MASTER EN SCIENCES INFORMATIQUES

Contribution à la mise au point d'un langage d'accès aux bases de données temporelles

RAMLOT, Olivier

Award date:
2000

FACULTES UNIVERSITAIRES NOTRE-DAME DE LA PAIX
INSTITUT D'INFORMATIQUE
NAMUR
(Belgique)

Contribution à la mise au point d'un langage d'accès aux bases de données temporelles

Olivier RAMLOT

Mémoire présenté en vue de l'obtention
du grade de Maître en Informatique.

Année académique 1999-2000

Résumé

Les bases de données temporelles sont des bases de données capables d'exprimer les aspects temporels du monde réel. Les langages classiques n'ont pas été conçus pour accéder à ce type de base. C'est donc pour cette raison qu'il est nécessaire de définir un nouveau langage spécifiquement élaboré pour les bases de données temporelles. Cet ouvrage décrit notre contribution à l'élaboration d'un tel langage et détaille la syntaxe et la sémantique de celui-ci . Ce nouveau langage mis au point permet notamment la réalisation de concepts temporels avancés tels que l'opérateur de synthèse et la jointure temporelle. De plus, de nouvelles fonctions de type ODBC (Open DataBase Connectivity, Microsoft) ont été élaborées de manière à permettre l'exécution de requêtes formulées au moyen de ce langage.

Abstract

Temporal databases are databases able to express temporal aspects of the real world. The classic languages weren't originally formed in order to access to this database type. Consequently a new language specifically construct for temporal databases was required. This work describes our contribution to the development of this kind of language and focuses on its syntax and its semantic. This new language accepts some complex temporal concepts like the coalescing operator and the temporal join. Furthermore, new ODBC (Open DataBase Connectivity, Microsoft) functions have been developed in order to allow the execution of queries written in this language.

Table des matières

1 Introduction générale 1

1.1 Prologue 1
1.2 Objectifs poursuivis 1
1.3 Méthode de travail 2
1.4 Remerciements 2

2 Bases de données temporelles : les concepts généraux 3

2.1 Introduction 3
2.2 Base de données non temporelle 3
2.3 Concept du temps dans les bases de données temporelles 5
2.4 Base de données mono-temporelle 5
2.5 Base de données bitemporelle 7
2.6 Lien entre les bases de données non temporelles, mono-temporelles et bitemporelles 9

3 Hypothèses et vocabulaire de travail 11

3.1 Introduction 11
3.2 Hypothèses de travail 11
3.3 Vocabulaire utilisé 14

4 Bases de données temporelles : explication des concepts avancés 15

4.1 Introduction 15
4.2 Opérateur de synthèse sur une base de données mono-temporelle 15
4.3 Opérateur de synthèse sur une base de données bitemporelle 17
4.4 Clé étrangère temporelle 26
4.5 Jointure temporelle 28

5 Optimisation 37

5.1 Introduction 37
5.2 Technique d'optimisation 37

6 Un langage de requêtes temporelles : le langage mini-TSQL2 41

6.1 Introduction 41
6.2 Caractéristiques d'un bon langage temporel 41
6.3 Le langage mini-TSQL2 42

6.3.1 Généralités 42
6.3.2 Sélection dans une seule table 43
6.3.3 Sélection dans deux tables 48
6.4 La syntaxe complète du langage mini-TSQL2 49
6.5 La sémantique du langage mini-TSQL2 50
6.5.1 Généralités 50
6.5.2 La clause `where` du langage mini-TSQL2 50
6.5.3 La clause `select` du langage mini-TSQL2 51
6.5.4 Interprétation d'une requête mini-TSQL2 53
6.6 Respect des desiderata des utilisateurs 59
6.7 Exemples de requêtes 60
6.8 Puissance du langage mini-TSQL2 62

7 Implémentation du langage temporel mini-TSQL2 67

7.1 Introduction 67
7.2 Implémentation d'un traducteur 68
7.2.1 Traducteur pour les requêtes accédant à une seule table 68
7.2.2 Traducteur pour les requêtes accédant à deux tables 75
7.3 Amélioration d'ODBC : TODBC 79
7.3.1 Pourquoi utiliser ODBC ? 79
7.3.2 Présentation sommaire d'ODBC 80
7.3.3 TODBC 81
7.3.4 Exemple de programme TODBC 87

8 Conclusions 93

8.1 Particularités du langage développé 93
8.2 Perspectives d'avenir 94

Bibliographie 95

1
Introduction générale

1.1 Prologue

L'homme, durant son histoire, a subi de nombreux bouleversements. Parmi ceux-ci, la maîtrise du temps, grâce à l'invention de l'horloge, est l'un des plus importants. En effet, l'homme qui rythmait jusque-là sa vie en fonction de la nature, a pris son indépendance en créant cette quatrième dimension. C'est maintenant au tour des informaticiens de vouloir la maîtriser en l'introduisant dans les bases de données. Poussés par un intérêt grandissant manifesté par les entreprises, les chercheurs ont donc créé des bases de données capables d'exprimer les aspects temporels du monde réel. C'est ce qu'ils appelleront les « bases de données temporelles ».

1.2 Objectifs poursuivis

Le but de ce travail est, dans un premier temps, d'examiner et de définir un nouveau langage qui permettra de simplifier l'accès aux bases de données temporelles. En effet, la faiblesse des langages actuels est bien réelle car ils n'ont pas été conçus au départ pour traiter les aspects temporels des événements enregistrés dans une base de données. A l'heure actuelle, les requêtes formulées en vue de récupérer des données temporelles sont complexes et difficilement réalisables. C'est donc pour cette raison qu'il est nécessaire de définir un nouveau langage spécifiquement élaboré pour les bases de données temporelles. Pour réaliser celui-ci, une extension du langage SQL qui est le langage le plus répandu dans le monde des bases de données sera étudiée. En fait, nous essayerons d'améliorer SQL pour qu'il puisse manipuler les différents aspects temporels.

La deuxième partie de ce travail consiste à créer un prototype d'implémentation qui permettra à tout programmeur de réaliser une requête écrite dans notre nouveau langage.

Il est important de souligner que le but premier de ce travail n'est pas de fournir une solution optimale mais bien d'éveiller le lecteur aux différentes possibilités ainsi qu'aux nombreux problèmes rencontrés lors de la définition et de l'implémentation d'un langage temporel.

1.3 Méthode de travail

Dans une première étape, nous nous attacherons à présenter et à décrire, selon un éclairage personnel, les différentes bases de données temporelles (chapitre 2). Ces descriptions nous fourniront une base théorique suffisante pour créer notre nouveau langage temporel.

Ensuite, nous définirons de manière précise les hypothèses de travail et plus particulièrement le type de bases de données temporelles sur lesquelles nous allons travailler (chapitre 3). En effet, comme les bases de données temporelles appartiennent toujours au domaine de la recherche, aucun standard n'existe à l'heure actuelle.

Dès que les différentes hypothèses de travail auront été identifiées, une explication détaillée des concepts temporels pourra être menée (chapitre 4). Celle-ci servira de base solide pour l'implémentation de notre langage temporel.

Afin d'accéder au mieux aux données d'une base temporelle, une technique d'optimisation a été élaborée par l'équipe du projet TimeStamp[1]. Celle-ci permet un accès plus rapide aux données temporelles. Comme cette optimisation a un impact sur l'implémentation de notre langage, un chapitre lui sera consacré (chapitre 5).

Enfin, nous présenterons la syntaxe et la sémantique de notre nouveau langage temporel (chapitre 6) ainsi qu'un prototype d'implémentation (chapitre 7).

Nous terminerons par des conclusions dans lesquelles nous développerons une série de perspectives d'avenir (chapitre 8).

1.4 Remerciements

Qu'il me soit permis d'exprimer toute ma gratitude à M. le Professeur J.L. HAINAUT qui n'a jamais cessé de m'accorder sa confiance et de me prodiguer ses encouragements tout au long de la réalisation du présent travail.

Je voudrais également témoigner ma reconnaissance à M. le Professeur B. THEODOULIDIS ainsi qu'à toute son équipe, grâce auxquels j'ai pu enrichir mes connaissances dans le domaine des bases de données temporelles durant mon séjour à Manchester (UK).

Mes remerciements s'adressent enfin à tous ceux qui ont contribué à la mise en œuvre de ce travail, tant sur le plan des idées que sur le plan matériel. Je pense principalement aux équipes des projets TimeStamp et DB-MAIN, et plus particulièrement à V. DETIENNE.

---

[1] Le Projet TimeStamp est financé par la Région wallonne (Contrat 9713563)

2
Bases de données temporelles : les concepts généraux

2.1 Introduction

Ce chapitre a pour but d'introduire et de définir les notions fondamentales nécessaires à l'étude des bases de données temporelles. Ce chapitre possède un caractère général et est indépendant de tout SGBD particulier. Avant d'aborder concrètement ce chapitre et pour clarifier notre raisonnement, un certain nombre de définitions préalables doivent être rappelées.

Une **base de données** modélise et enregistre des informations sur une partie de la réalité, dénommée « mini-monde » par certains auteurs tels que [JENSEN, 99]. Actuellement, il existe deux grands types de bases de données : les bases de données relationnelles et les bases de données orientées objets.

Dans une **base de données relationnelle**, les différents aspects du « mini-monde » sont représentés grâce aux concepts mathématiques de relation et de domaine alors qu'une **base de données orientée objets** utilise uniquement le concept d'objet. Dans ce travail, lorsque nous omettons de le spécifier, il faut considérer que la base de données est de type relationnel.

Ce chapitre développe et définit la notion de **base de données temporelle**. Ce type de base de données existe sous deux formes : les **bases de données mono-temporelles** (paragraphe 2.3) et **bitemporelles** (paragraphe 2.4). Cependant, afin de bien comprendre la différence existant entre les bases de données **temporelles** et **non temporelles**, nous rappelons la signification d'une base de données **non temporelle** (paragraphe 2.2).

2.2 Base de données non temporelle

Une base de données non temporelle est une base de données représentant notre connaissance **actuelle** du « mini-monde ». Dans une base de données non temporelle, les enregistrements ou faits se trouvant dans les différentes tables expriment ce que nous savons aujourd'hui à propos du « mini-monde ». La grande majorité des bases de données, jusqu'à ce jour, sont non temporelles.

La table `ELEVE_1` (figure 2.1) est un exemple de table non temporelle. Celle-ci contient des informations concernant les élèves d'un lycée. Tous les enregistrements de cette table peuvent être représentés sur une ligne du temps par un point (figure 2.2).

ELEVE_1			
NUMERO	**NOM**	**PRENOM**	**SECTION**
0	Dupuis	Alex	Histoire
1	Paillo	Emile	Histoire
2	Dupond	Léopold	Info

Figure 2.1: Exemple d'enregistrements de la table non temporelle ELEVE_1.

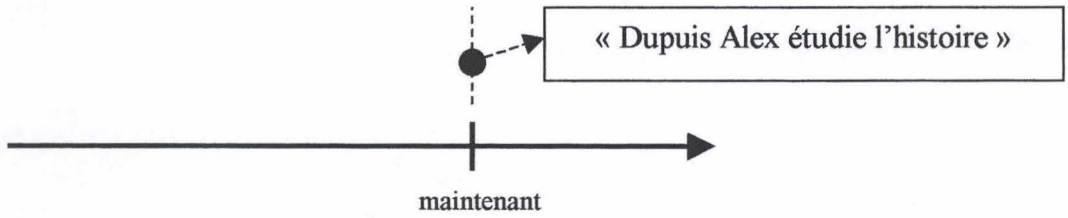


Figure 2.2: Représentation d'un enregistrement non temporel de la table ELEVE_1 sur une ligne du temps.

Tous les enregistrements de la table ELEVE_1 expriment l'ensemble des connaissances que nous avons actuellement à propos des élèves. Toutefois, la durée de nos connaissances n'est pas connue. En conséquence, nous n'avons aucune information nous permettant de dire si cela fait 10 secondes ou 1 an que nous savons que Dupuis Alex est en histoire ni même s'il y restera longtemps. Notre seule certitude est qu'il y est pour le moment. C'est pour cette raison qu'en anglais une base de données non temporelle est appelée un « **snapshot** » (un instantané, en français).

Dans le domaine des bases de données non temporelles et temporelles, le vocabulaire utilisé a beaucoup d'importance. En effet, dans les bases de données non temporelles, nous avons mélangé les termes de connaissance et d'enregistrement. En fait, nous avons tout simplement pris comme hypothèse implicite que toutes nos connaissances actuelles concernant le mini-monde sont enregistrées dans la base de données. Cela n'est pas toujours le cas dans la réalité, comme nous le soulignerons dans le chapitre sur les bases de données temporelles. En effet, nous pouvons très bien connaître un fait sans pour autant l'avoir déjà enregistré dans la base de données.

Ce n'est pas parce que nous qualifions une base de données de non temporelle que celle-ci est dépourvue de tout caractère temporel. En effet, nous pouvons trouver, dans une table non temporelle, un attribut dont le domaine est une date, une heure... Ce type d'attribut, appelé donnée temporelle (attribut temporel), doit être considéré comme tout autre attribut de l'enregistrement.

2.3 Concept du temps dans les bases de données temporelles

Deux types de temps distincts existent dans le domaine des bases de données temporelles : le « valid time » et le « transaction time ». C'est à travers ces deux concepts que la différence entre une connaissance et un enregistrement prend toute sa signification.

Une base de donnée qui manipule le « **valid time** » possède pour chaque enregistrement une période de validité. C'est la période durant laquelle un enregistrement (ou fait) est véridique dans le « mini-monde ». La période de validité touche à la connaissance des faits et pas à leur enregistrement. Par exemple, nous pourrions avoir une période de validité s'étendant du 01/10/1998 au 15/12/1998 pour le fait « Dupuis Alex étudie l'histoire ». Cela signifierait que dans « le mini-monde » à modéliser, Dupuis Alex a bel et bien étudié l'histoire entre les dates du 01/10/1998 et le 15/12/1998. Cependant, cette période de validité ne signifie en aucun cas que le 1/10/1998 ce fait ait été effectivement enregistré dans notre base de donnée ni même que celui-ci ait été retiré le 15/12/1998. Nous pouvons donc dire que le « valid time » capture ***la variation des états du mini-monde*** [JENSEN, 99].

Une base de données qui manipule le « **transaction time** » possède pour chaque enregistrement une période de transaction. Cette période de transaction capture le moment où un fait est courant dans la base de données, c'est-à-dire l'instant où nous l'avons effectivement enregistré et supprimé de la base de données. La période de transaction permet de capturer ***la variation des états de la base de données***. Contrairement à la période de validité, la période de transaction peut être générée automatiquement par le SGBD.

Afin d'indiquer dans une base de données temporelle qu'un état est courant et pour préserver le concept de période, nous utiliserons pour le « valid time » et le « transaction time » une date de fin finie mais inatteignable, par exemple 01/01/3000. Sémantiquement, cette date permet d'exprimer la situation d'un enregistrement courant dont sa date de fin n'est pas encore connue.

2.4 Base de données mono-temporelle

Une base de données mono-temporelle ne manipule qu'un seul type de temps : soit le « valid time » ou soit le « transaction time ». Nous dirons que cette base de données possède une et une seule dimension temporelle.

Une base de données mono-temporelle est une base de données représentant notre connaissance actuelle aussi bien du passé, du présent que du futur. Le futur n'existe cependant pas pour les bases de données avec « transaction time ». En effet, il n'est pas possible de prévoir les états futurs de la base de données. Néanmoins, cette affirmation doit être nuancée. En effet, grâce à la date particulière du 01/01/3000, les états futurs qui seraient identiques à l'état courant sont

représentés. Les bases de données mono-temporelles dont les tables représentent seulement les états courants et passés sont appelées des **historiques d'états**.

La table ELEVE_2 (figure 2.3) modélise notre connaissance actuelle de l'évolution dans le temps des sections de chacun des étudiants. Par exemple, nous pouvons affirmer que Renard Max a étudié l'économie du 17/03/99 au 30/03/99 et qu'ensuite il a changé de section pour le droit. Tous les changements d'étude d'un élève au cours du temps peuvent être représentés sur une droite (figure 2.4).

ELEVE_2					
NUMERO	**NOM**	**PRENOM**	**SECTION***	**DEBUT**	**FIN**
0	Dupuis	Alex	Droit	10/03/1999	20/03/1999
0	Dupuis	Alex	Eco	20/03/1999	03/04/1999
0	Dupuis	Alex	Histoire	03/04/1999	01/01/3000
...	...	...	...	...	...
3	Renard	Max	Eco	17/03/1999	30/03/1999
3	Renard	Max	Droit	30/03/1999	09/04/1999
4	Tondu	Jeremy	Histoire	21/04/1999	28/04/1999
4	Tondu	Jeremy	Eco	28/04/1999	03/05/1999
....	....	....	....	....	.....

Figure 2.3: Exemple d'enregistrements de la table mono-temporelle ELEVE_2 (historique d'états).

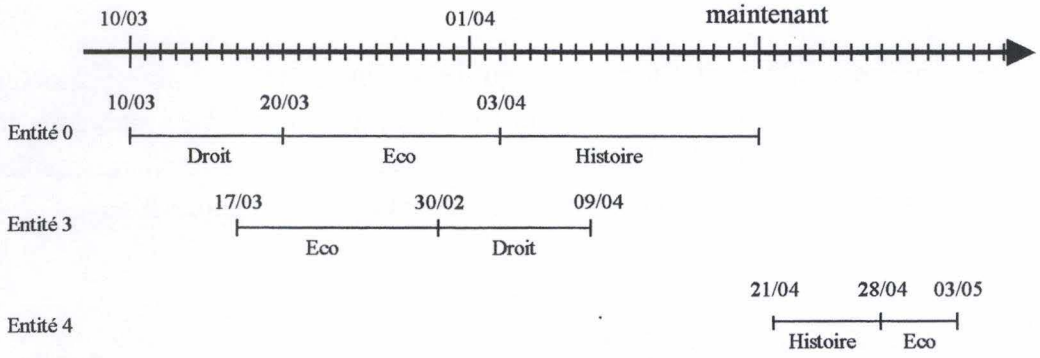


Figure 2.4: Représentation graphique de la table mono-temporelle ELEVE_2 à la date du 21/04/99.

Pour pouvoir travailler dans le monde des bases de données mono-temporelles, un certain nombre d'éléments doivent être précisés.

Un **état courant (C**ourant**)** est un état dont la période de validité ou la période de transaction comprend le moment présent. Une table ne possédant que des états courants n'est rien d'autre qu'une table non temporelle.

Un **état passé (Historique)** est un état qui n'est pas courant et dont la période de validité (période de transaction) possède une valeur de fin antérieure au moment présent.

Un **état futur (Futur)** est un état dont la période de validité possède une valeur de début postérieure au moment présent. Une base de données mono-temporelle avec « transaction time » ne peut pas enregistrer d'états futurs qui soient différents des états courants. Cela serait, en effet, contraire à la définition du « transaction time ».

Une **entité** est un élément dont nous souhaitons tracer la variation dans le temps. Cet élément est identifié par l'identifiant de la table non temporelle dérivé de la table mono-temporelle en ne gardant que les états courants. Par exemple, dans la table précédente, une entité représente un étudiant et est identifiée par son numéro.

Si I est l'**identifiant** d'une entité, alors `{I,DEBUT}` et `{I,FIN}` sont deux identifiants de l'historique correspondant. L'identifiant d'une entité d'une table mono-temporelle doit être stable et non recyclable . En effet, pour pouvoir retracer l'historique d'une entité, il est interdit de modifier son identifiant, ni même de le réutiliser pour une autre entité.

2.5 Base de données bitemporelle

Les bases de données bitemporelles possèdent la structure la plus riche car elles manipulent les deux types de temps : le « valid time » et le « transaction time ». Ce type de base de données possède donc deux dimensions temporelles.

Les bases de données bitemporelles sont apparues pour combler un manque d'expressivité des bases de données non temporelles et mono-temporelles. En effet, jusqu'à présent, nous ne pouvions enregistrer que notre connaissance actuelle du moment présent (bases de données non temporelles) ou du présent, passé et futur (bases de données mono-temporelles). Nous n'étions cependant pas capable de savoir ce que nous connaissions il y a 2 jours ou il y a 10 ans. C'est ce que permettent les bases de données bitemporelles en enregistrant notre **connaissance passée et actuelle** à propos d'évènements passés, présents et futurs du « mini-monde ». Nous pouvons donc dire qu'une telle base modélise non seulement ce que nous connaissons aujourd'hui mais aussi ce que nous connaissions à un instant précis dans le passé. Pour obtenir de telles bases de données, nous allons ajouter à chaque enregistrement d'une base de données mono-temporelles de type « valid time » une période de transaction. Celle-ci exprimera la durée pendant laquelle nous avons cru ou nous croyons que cet enregistrement avec une période de validité déterminée était véridique. Pour une meilleure compréhension, [HAINAUT, 99] nous propose la représentation ci-dessous (figure 2.5).

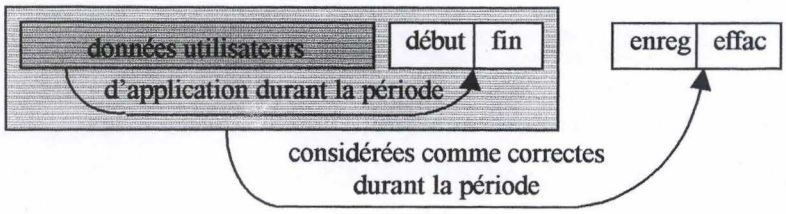


Figure 2.5: Rôles des périodes temporelles associées aux données utilisateurs.

ELEVE_3							
NUMERO	**NOM**	**PRENOM**	**SECTION***	**V-DEBUT**	**V-FIN**	**T-DEBUT**	**T-FIN**
0	Dupuis	Alex	Droit	10/03/1999	01/01/3000	11/03/99	23/03/1999
0	Dupuis	Alex	Droit	10/03/1999	20/03/1999	23/03/99	01/01/3000
0	Dupuis	Alex	Eco	20/03/1999	01/01/3000	23/03/99	04/04/1999
0	Dupuis	Alex	Eco	20/03/1999	03/04/1999	04/04/99	01/01/3000
0	Dupuis	Alex	Histoire	03/04/1999	01/01/3000	04/04/99	01/01/3000
...	...	...	...	...	...	....	....
3	Renard	Max	Eco	17/03/1999	01/01/3000	18/03/99	02/04/1999
3	Renard	Max	Eco	17/03/1999	30/03/1999	02/04/99	01/01/3000
3	Renard	Max	Droit	30/03/1999	01/01/3000	02/04/99	05/04/1999
3	Renard	Max	Droit	30/02/1999	04/04/1999	05/04/99	01/01/3000
...	....	...	...	...	...	...	...

Figure 2.6: Exemple d'enregistrements de la table bitemporelle ELEVE_3.

Pour l'entité 0, la table `ELEVE_3` (figure 2.6) nous informe, par exemple, qu'entre le 11/03 et le 23/03, nous étions au courant que Dupuis Alex étudiait le droit depuis le 10/03 ou que depuis le 04/04, nous savons qu'il a étudié l'économie entre le 20/03 et le 03/04.

Pour chaque entité, nous pouvons représenter la situation de nos connaissances dans le temps grâce à un graphique à deux dimensions (« valid time » et « transaction time »). Sur un tel graphique, chaque enregistrement, grâce aux dates de début et de fin du « valid time » et du « transaction time », forme un rectangle. Le graphique de l'entité 0 de la table `ELEVE_3` en est un exemple (figure 2.7).

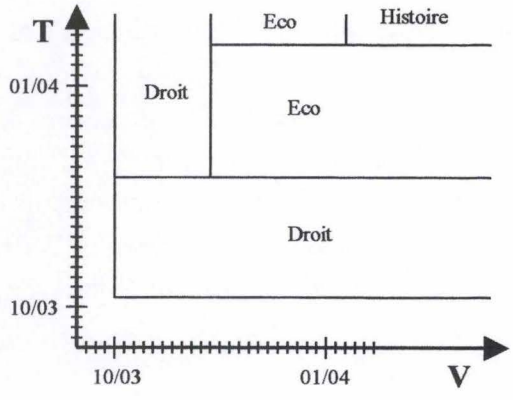


Figure 2.7: Représentation graphique de l'historique de l'entité 0 de la table bitemporelle `ELEVE_3`.

Une remarque importante doit pourtant être formulée. Pour pouvoir obtenir une base de données bitemporelle, nous avons dû modifier le concept du « transaction time ». Jusqu'à présent, la période de transaction avait pour but de représenter la période durant laquelle un état avait été ou est considéré comme courant pour une entité. Maintenant, cette période permet de capturer les instants où l'état est considéré comme correct.

Comme pour les bases de données mono-temporelles, différents éléments doivent être définis.

Un **état courant** (Courant) est un état dont les périodes de validité et de transaction comprennent le moment présent.

Un **état passé valide** (Historique) est un état dont la période de validité est antérieure au moment présent et dont la période de transaction comprend le moment présent.

Un **état futur** (Futur) est un état dont la période de validité est postérieure au moment présent et dont la période de transaction comprend le moment présent.

Un **état non valide** (Non Valide) est un état dont la période de transaction ne comprend pas le moment présent.

Si I est l'**identifiant** d'une entité, alors `{I, V_DEBUT, T_DEBUT}` est un identifiant de la table bitemporelle correspondante.

2.6 Lien entre les bases de données non temporelles, mono-temporelles et bitemporelles

Pour bien comprendre le lien existant entre une base de données non temporelle, mono-temporelle et bitemporelle, nous allons essayer de déterminer, à partir d'une base de données bitemporelle, l'ensemble des informations qui font partie respectivement d'une base de données mono-temporelle ou non temporelle. Pour nous éclairer, nous utiliserons la table `ELEVE_3`.

Si nous devions dériver de la table `ELEVE_3` notre connaissance actuelle de la variation des états du mini-monde qui est une base de données mono-temporelle avec « valid time » nous effectuerions la requête suivante :

```
select NUMERO,NOM,PRENOM,SECTION,V_DEBUT,V_FIN
from   ELEVE_3
where  T_FIN = '01/01/3000'
```

De la même manière, pour dériver l'ensemble des données véridiques actuellement, qui est une base de données non temporelle, nous utiliserions la requête suivante :

```
select NUMERO, NOM, PRENOM, SECTION
from ELEVE_3
where T_FIN = '01/01/3000' et V_FIN = '01/01/3000'
```

Cependant, cette requête n'est correcte que si nous travaillons sans états futurs. Dans le cas contraire, nous devrions utiliser la requête suivante où « maintenant » doit être remplacé par la date d'aujourd'hui:

```
select NUMERO, NOM, PRENOM, SECTION
from ELEVE_3
where T_FIN = '01/01/3000' and V_DEBUT <= maintenant and V_FIN > maintenant
```

Comme le « transaction time » d'une base de données bitemporelle n'a pas la même signification que celui d'une base de données mono-temporelle, il n'est pas possible de dériver l'un à partir de l'autre.

3
Hypothèses et vocabulaire de travail

3.1 Introduction

Dans le domaine des bases de données temporelles, chaque informaticien ou chercheur travaille dans des conditions et des hypothèses qui lui sont propres. En effet, comme dans tout domaine de recherche, les standards et les normes n'existent pas ou très peu. C'est donc pour une raison de précision et pour permettre au lecteur une pleine compréhension de notre travail que nous nous devons à notre tour d'énoncer les hypothèses (paragraphe 3.2) que nous avons posées ainsi que de définir le vocabulaire utilisé (paragraphe 3.3).

3.2 Hypothèses de travail

L'ensemble de ces hypothèses a été déterminé dans le cadre du projet TimeStamp.

- Nous allons toujours travailler sur des bases de données relationnelles en 1FN (première forme normale).

- Bien que plusieurs types d'historiques existent (historiques d'entités, historiques d'associations, historiques dérivant d'une requête), nous nous concentrerons uniquement sur les historiques d'entités.

- Pour des raisons de facilité et de lisibilité, nous représenterons les dates par des entiers.

- Les périodes de validité et de transaction sont représentées par un intervalle de deux dates fermé à gauche et ouvert à droite (par exemple: [10,38[). Chacune de celles-ci est respectivement appelée date de début et date de fin.

- Pour les états courants, nous utiliserons le nombre 999 comme date de fin finie mais inatteignable.

- Pour qu'une période soit significative, il faut que quelque soit l'état de l'historique, sa date de début soit toujours antérieure à sa date de fin.

- Les valeurs des colonnes d'une table temporelle dont on veut tracer la variation à travers le temps ne peuvent pas être nulles.

Base de données mono-temporelle

- A chaque instant, pour chaque entité, il ne peut y avoir qu'au plus un seul enregistrement.
  Exemple: « A un instant précis, un étudiant ne peut être que dans, au plus, une seule section ».

- Au sein de la variation des états d'une entité, il faut qu'il y ait continuité des périodes.
  Exemple: « Un étudiant qui quitte une section, doit soit être réinscrit directement dans une nouvelle section, soit ne plus jamais étudier ».

- Nous n'accepterons pas d'états consécutifs véhiculant la même information.
  Exemple: « Il serait incorrect, par exemple, d'avoir dans notre base de données un enregistrement véhiculant les informations sur les études en histoire de Dupuis Alex du 10/03 au 12/03 et un autre fournissant exactement les mêmes informations pour les dates du 12/03 au 14/03. En effet, dans ce cas un seul enregistrement s'étalant du 10/03 au 14/03 aurait dû être utilisé ».

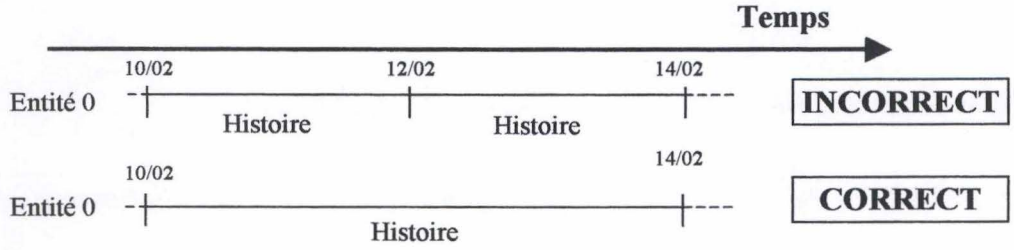


Figure 3.1: Hypothèse de travail: « interdiction d'avoir des états consécutifs véhiculant la même information ».

- Pour des raisons de conventions, si I est l'identifiant d'entité d'une table mono-temporelle, `{I, DEBUT}` en sera l'identifiant primaire.

Base données bitemporelle

- Comme le but principal des bases de données bitemporelles est de pouvoir déterminer exactement notre connaissance dans le passé, il nous est **interdit de modifier les informations d'un enregistrement stocké dans la table** au risque de ne plus pouvoir retracer notre connaissance passée. En effet, si nous voulons changer la valeur d'un enregistrement, un nouveau doit être créé en y ajustant judicieusement les différentes périodes.

- A chaque instant de transaction (période de transaction), au sein de la variation des états d'une entité connus à ce moment, il faut qu'il y ait continuité des différentes périodes de validité.
  Exemple: « Pour l'entité 0, au sein de la variation des états connus le 24/03, les différentes périodes de validité sont continues ».

- A chaque instant de la réalité (période de validité), au sein de la variation de mes croyances pour cet instant, il faut qu'il y ait continuité des différentes périodes de transaction.
  Exemple: « Pour l'entité 0, au sein de la variation des états de mes croyances pour le 24/03, les différentes périodes de transaction sont continues ».

- Ces deux dernières hypothèses peuvent être exprimées graphiquement. Elles signifient que le rectangle représentant la variation des états d'une entité dans les deux dimensions ne possèdent pas de trou.
  Exemple: « Le rectangle représentant nos connaissances sur l'entité 0 (figure 2.7) ne possède pas de trou ».

- Nous n'accepterons pas d'états consécutifs, en termes de « valid time », véhiculant la même information. Plus concrètement, pour tout instant t (t appartenant au « transaction time »), l'historique observé à cet instant, ne possèdera pas deux enregistrements consécutifs identiques.
  Exemple : Il serait incorrect d'avoir dans notre base de données les enregistrements suivants:
  - du 20/03 au 28/03,.nous avons cru que Dupuis Alex étudiait l'histoire du 10/03 au 18/03,
  - du 24/03 au 02/04, nous avons cru que Dupuis Alex étudiait l'histoire du 18/03 au 22/03.

  Le cas correct aurait été de posséder les enregistrement suivants:
  - du 20/03 au 24/03, nous avons cru que Dupuis Alex étudiait l'histoire du 10/03 au 18/03,
  - du 24/03 au 28/03, nous avons cru Dupuis Alex avait étudiait l'histoire du 10/03 au 22/03,
  - du 28/03 au 02/04, nous avons cru Dupuis Alex avait étudiait l'histoire du 18/03 au 22/03.

  Graphiquement :

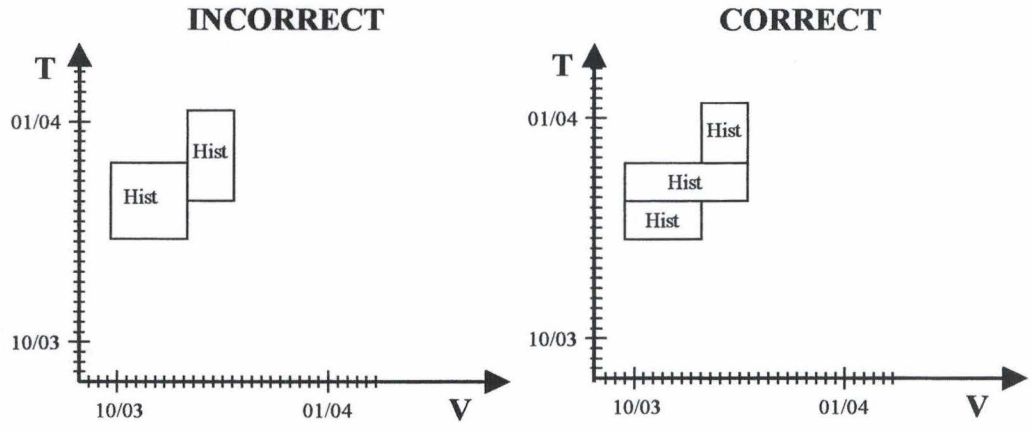


Figure 3.2: Hypothèse de travail : « à tout instant du « transaction time », il n'existe pas deux enregistrements consécutifs en terme de « valid time » véhiculant la même information ».

Cette même constatation ne peut, cependant, pas être faite sur l'axe du « valid time ». En d'autres termes, pour tout v (v appartenant au « valid time »), l'historique de mes croyances à l'instant v peut posséder deux enregistrements consécutifs véhiculant la même information. Nous pouvons facilement le vérifier sur l'exemple correct précédent.

- Une contrainte particulière est, cependant, assignée à l'axe du « valid time » en coordination avec le «transaction time». Dans une table bitemporelle, il est impossible de trouver deux enregistrements véhiculant la même information, dont les «transaction time» sont consécutifs et de même « valid time ».

Graphiquement :

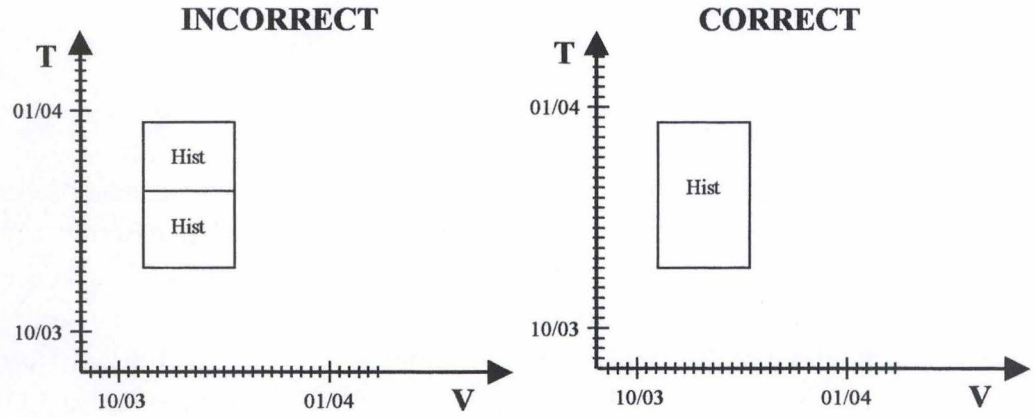


Figure 3.3: Hypothèse de travail : « il n'existe pas deux enregistrements véhiculant la même information, dont les «transaction time» sont consécutifs et de même « valid time » ».

- Pour des raisons de conventions, si I est l'identifiant d'une entité d'une table bitemporelle, `{I, V_DEBUT, T_DEBUT}` en sera l'identifiant primaire.

3.3 Vocabulaire utilisé

- Une **table temporelle** est une table possédant au moins une colonne temporelle.
  Exemple : `Eleve_3`.

- Une **table temporelle normalisée** est une table temporelle respectant l'ensemble des hypothèses de travail dans lesquelles nous évoluons.

- Une **table temporelle non normalisée** est une table temporelle ne respectant pas l'ensemble des hypothèses de travail dans lesquelles nous évoluons.

- Une **colonne temporelle** est une colonne d'une table temporelle dont nous voulons tracer la variation à travers le temps.
  Exemple : la colonne `SECTION` de la table `Eleve_3`.
  Notation : dans une table temporelle, une colonne temporelle sera annotée du signe « * ».

- Une **colonne non temporelle** est une colonne dont nous ne désirons pas tracer la variation à travers le temps.
  Exemple : la colonne `NOM` de la table `Eleve_3`.

4
Bases de données temporelles : explication des concepts avancés

4.1 Introduction

Ce chapitre met en évidence les différentes réflexions et travaux que nous avons menés sur certains concepts complexes des bases de données temporelles. Ces concepts sont spécifiquement étudiés en tenant compte de nos hypothèses de travail préalablement citées. Ce chapitre est important car il forme l'ossature de l'implémentation de notre langage temporel.

Ce chapitre aborde plus particulièrement le concept d'opérateur de synthèse sur des bases de données mono-temporelles (paragraphe 4.2) et bitemporelles (paragraphe 4.3). Par la suite, une étude sur la clé étrangère (paragraphe 4.4) et sur la jointure temporelle (paragraphe 4.5) est menée en vue d'éclairer les relations existant entre deux tables temporelles.

4.2 Opérateur de synthèse sur une base de données mono-temporelle

L'une des difficultés majeures des bases de données temporelles réside dans la compréhension de l'opérateur de synthèse appelé **coalescing**. Cet opérateur est utile lorsqu'on travaille sur une table temporelle possédant au moins deux colonnes temporelles.

Considérons la table PERSONNE_1 (figure 4.1) possédant deux colonnes temporelles: ADRESSE et SALAIRE. Si nous désirons obtenir uniquement les informations concernant le changement d'adresse des différentes personnes, la projection sur les colonnes NOM, ADRESSE, V_DEBUT et V_FIN ne nous satisfait pas (figure 4.1, PERSONNE_2). En effet, nous pouvons remarquer qu'il existe un certain nombre d'états consécutifs identiques, ce qui est contraire à nos hypothèses de travail. Pour obtenir une table résultat conforme à celles-ci (figure 4.1, PERSONNE_3), il faut que nous puissions construire cette dernière à partir de chaque suite d'états consécutifs de même valeur d'ADRESSE. C'est le rôle de l'opérateur de synthèse.

Il est possible de construire cet opérateur sous la forme d'une séquence de requêtes SQL compliquées. Néanmoins, une autre solution est émise dans [HAINAUT, 99]. Celle-ci consiste à trier la table selon l'identifiant conventionnel pour une table mono-temporelle, ici NOM et V_DEBUT, pour ensuite, grâce à un langage de 3ème génération, parcourir les différents états pour les synthétiser. En effet, les enregistrements qui se suivent et qui possèdent les mêmes valeurs de NOM et ADRESSE seront synthétisés. C'est cette solution que nous mettrons en œuvre dans notre langage

temporel. L'intérêt majeur de cette solution est qu'un seul passage dans la table suffit pour réaliser le coalescing.

PERSONNE_1				
NOM	**ADRESSE***	**SALAIRE***	**V_DEBUT**	**V_FIN**
Léo	Liège	50000	10	35
Léo	Bruxelles	100000	35	999
René	Namur	20000	15	25
René	Namur	30000	25	50
René	Namur	50000	50	75
René	Bruxelles	50000	75	90
Jean	Liége	25000	20	25
Jean	Namur	25000	25	40
Jean	Namur	60000	40	999

PERSONNE_2			
NOM	**ADRESSE***	**V_DEBUT***	**V_FIN**
Léo	Liège	10	35
Léo	Bruxelles	35	999
René	Namur	15	25
René	Namur	25	50
René	Namur	50	75
René	Bruxelles	75	90
Jean	Liége	20	25
Jean	Namur	25	40
Jean	Namur	40	999

PERSONNE_3			
NOM	**ADRESSE***	**V_DEBUT**	**V_FIN**
Léo	Liège	10	35
Léo	Bruxelles	35	999
René	Namur	15	75
René	Bruxelles	75	90
Jean	Liége	20	25
Jean	Namur	25	999

Figure 4.1: Exemples d'enregistrements de tables illustrant l'action du coalescing pour une base de données mono-temporelle : `PERSONNE_1`, `PERSONNE_2`, `PERSONNE_3`.

L'implémentation de l'algorithme de coalescing pour une base de données mono-temporelle se déroule en deux étapes. Premièrement, la projection désirée ainsi que l'ordonnancement du résultat obtenu selon l'identifiant conventionnel (`{I, DEBUT}`) seront réalisés. Ensuite l'algorithme de la figure 4.2 sera appliqué.

Tant qu'il existe un enregistrement C à traiter **faire** :
{
Si C est le premier enregistrement
Alors C est stocké directement dans l'enregistrement de travail T
Sinon Si C possède exactement les mêmes valeurs d'attributs que T.
Alors C et T sont synthétisés
Sinon - T est inséré dans la table résultat R
- C est stocké dans T
Passer à l'enregistrement C suivant
}
Si T n'est pas vide
Alors T est inséré dans R

Figure 4.2: Algorithme de coalescing pour une table mono-temporelle.

Cet algorithme utilise un enregistrement de travail « T » qui joue le rôle de support sur lequel un enregistrement correct ou enregistrement final est fabriqué. Concrètement, la synthèse de deux enregistrements consiste à fusionner les deux périodes. Dans le cadre de cet algorithme, puisque les enregistrements sont ordonnés, seule la valeur de l'attribut `FIN` de l'enregistrement de travail est remplacé par la valeur de l'attribut `FIN` de l'enregistrement courant.

4.3 Opérateur de synthèse sur une base de données bitemporelle

L'opérateur de synthèse existe aussi sur les bases de données bitemporelles. Il permet lors d'une projection d'une table bitemporelle comportant au moins deux colonnes temporelles d'obtenir une table bitemporelle résultante correcte. Nous entendons par table bitemporelle correcte, une table respectant les hypothèses de travail énoncées au chapitre précédent.

Considérons la table bitemporelle de la figure 4.3. Pour une meilleure lisibilité de notre exposé, nous nous sommes limité aux enregistrements d'une seule entité, l'entité Léo.

PERSONNE_4						
NOM	**ADRESSE***	**SALAIRE***	**V_DEBUT**	**V_FIN**	**T_DEBUT**	**T_FIN**
Léo	Namur	50000	10	999	20	30
Léo	Namur	60000	10	20	30	45
Léo	Liège	60000	20	25	30	40
Léo	Liège	70000	25	999	30	40
Léo	Namur	80000	20	30	40	55
Léo	Bruxelles	80000	30	999	40	50
Léo	Namur	70000	10	20	45	55
Léo	Bruxelles	100000	30	999	50	60
Léo	Liège	50000	10	30	55	60
Léo	Liège	50000	10	35	60	999
Léo	Bruxelles	100000	35	999	60	999
...	...	...	...	...	...	...

Figure 4.3: Exemples d'enregistrements de l'entité Léo de la table bitemporelle : `PERSONNE_4`.

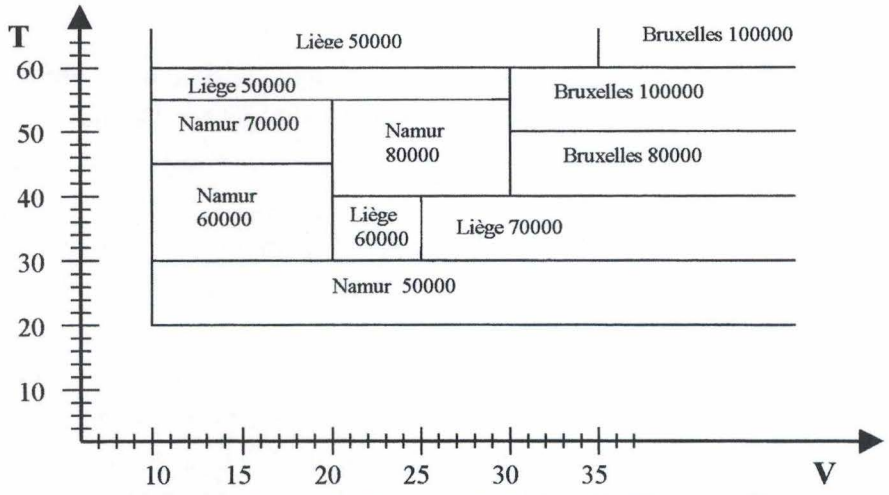


Figure 4.4: Représentation graphique de l'entité Léo de la table bitemporelle `PERSONNE_4`.

Pour faciliter la compréhension du coalescing sur une base de données bitemporelle, nous allons travailler principalement sur base des graphiques. Ceci nous est permis grâce à la bijection existant entre le graphique et la table dont il en est extrait. Le graphique de la table `PERSONNE_4` est représenté à la figure 4.4. Comme pour les bases de données mono-temporelles, si nous nous limitons à faire une simple projection, un certain nombre d'hypothèses de travail sont violées. En fait, deux violations d'hypothèses apparaissent. Il s'agit pour la première de l'interdiction d'avoir des enregistrements véhiculant la même information et jointifs horizontalement (selon le « valid time »). Quant à la deuxième, elle consiste en l'interdiction d'avoir des enregistrements véhiculant la même information et totalement jointifs verticalement (selon le « transaction time »). Deux enregistrements sont totalement jointifs lorsqu'ils partagent de manière complète un côté. Le graphique de la figure 4.5 énumère les violations rencontrées lorsqu'une requête de sélection des adresses est réalisée sur la table `PERSONNE_4`. Les deux types de problèmes engendrés par la violation des hypothèses de travail seront appelés respectivement : problème horizontal et problème vertical. Cette dénomination a été choisie pour mettre en évidence le sens dans lequel le problème est rencontré.

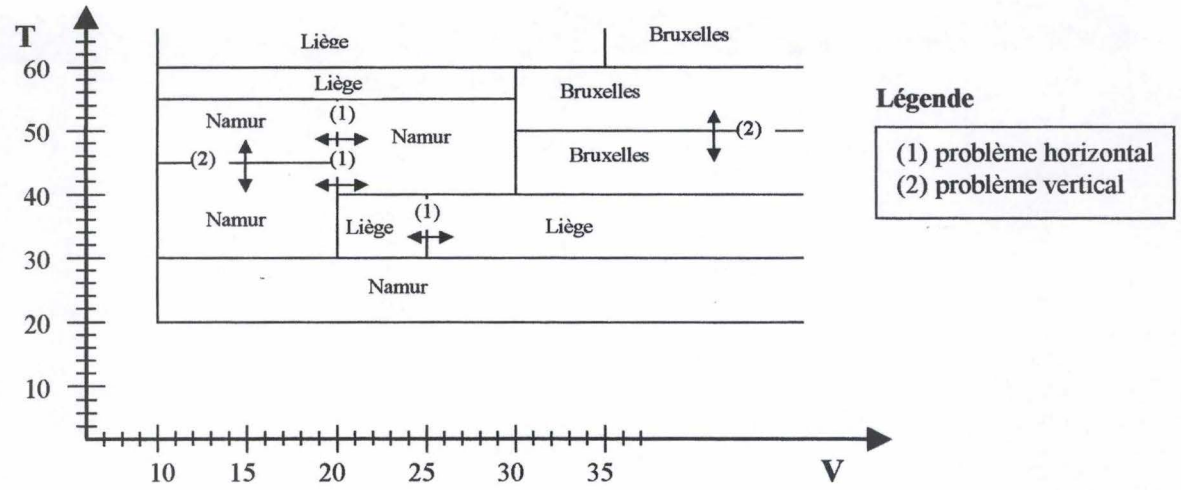


Figure 4.5: Représentation des problèmes rencontrés lors d'une projection d'une base bitemporelle.

Comme pour une base de données mono-temporelle, l'implémentation d'un algorithme de coalescing est requis. Comme celui-ci est complexe, l'ensemble du processus que nous avons élaboré sera détaillé. Tout d'abord, comme pour le cas mono-temporel, la projection désirée ainsi que l'ordonnancement du résultat obtenu selon l'identifiant `{I, T_DEBUT, V_DEBUT}` sera réalisé. Ensuite, un algorithme de colaescing bitemporel sera appliqué. Cet algorithme va traiter chaque entité l'une après l'autre. Grâce à l'ordonnancement, nous pouvons dire que tous les enregistrements relatifs à une entité sont ordonnés selon les attributs `T_DEBUT` et `V_DEBUT`. Cela signifie que d'un point de vue graphique, nous allons traiter les rectangles représentants les différents enregistrements de bas en haut et de gauche à droite. Notre algorithme est composé de deux phases distinctes. La première phase consistera à résoudre tous les problèmes horizontaux tandis que la deuxième s'attaquera spécifiquement aux problèmes verticaux.

La résolution des problèmes horizontaux de la **première phase** consiste à vérifier graphiquement si deux rectangles contigus par un de leurs côtés latéraux ne véhiculent pas les mêmes informations. Dans le petit exemple suivant (figure 4.6), grâce à l'ordonnancement, les rectangles seront traités dans l'ordre de leurs numéros.

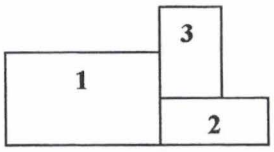


Figure 4.6: Exemple de situation graphique d'enregistrements jointifs.

Puisque les enregistrements 1 et 3 peuvent posséder les mêmes informations, nous devons absolument conserver le rectangle 1 une fois que celui-ci aura été traité. C'est pour cette raison que nous allons travailler avec non plus un enregistrement de travail comme dans le cas mono-temporel mais avec un tableau de travail dans lequel nous pourrons y stocker l'ensemble des enregistrements susceptibles d'être encore contigus à un autre rectangle. La question cruciale est de savoir à partir de quel moment un rectangle stocké dans la table de travail est totalement traité. Grâce à l'ordonnancement, nous pouvons répondre facilement à cette question. Le cas apparaît lorsque le `T_DEBUT` du rectangle courant est supérieur ou égal au `T_FIN` du rectangle stocké (la hauteur d'un rectangle est déterminée par `T_DEBUT` et `T_FIN`). En effet, à partir de ce moment, nous pouvons affirmer que tous les rectangles à venir ne seront plus jamais contigus latéralement ou horizontalement au rectangle stocké. L'algorithme de la première phase est représenté à la figure 4.7.

```
Tant qu'il existe un enregistrement C de l'entité courante E à traiter faire :
 {
 Si la table de travail T est vide
 Alors C est inséré directement dans T
 Sinon  /* C est inséré dans T qui est non vide*/
  Tant qu'il existe un enregistrement C' dans T à traiter et
        que C n'est pas encore synthétisé faire
  {
   Si C possède exactement les mêmes valeurs d'attribut que C' et
      si C et C' sont latéralement contigu
   Alors C et C' sont synthétisés
   Passer à l'enregistrement C' de T suivant
  }
 Si C n'a pas été synthétisé
 Alors C est inséré dans T
 /*il se peut que certains éléments n'aient pas été synthétisés*/
 Sinon  T doit être réexaminé pour vérifier que toutes les synthèses ont bien eu lieu
        (voir le problème de la tenaille)
 Tous les enregistrements de T qui ne sont plus susceptibles d'être contigus latéralement sont
  insérés dans la table résultat R.
  Passer à l'enregistrement C suivant
 }
Si T n'est pas vide
   Alors le contenu de T est inséré dans R
```

Figure 4.7: Algorithme de la phase 1.

Avant de poursuivre, plusieurs éléments de cet algorithme doivent être précisés. Parmi ceux-ci, le phénomène de synthèse et le problème de la tenaille sont les plus importants.

Le problème de la synthèse d'une table bitemporelle réside dans les multiples possibilités d'avoir deux rectangles ou enregistrements contigus latéralement. C'est pour cette raison que nous allons détailler minutieusement le procédé de synthèse. La figure 4.8 illustre, graphiquement, un des multiples cas de synthèse de deux enregistrements.

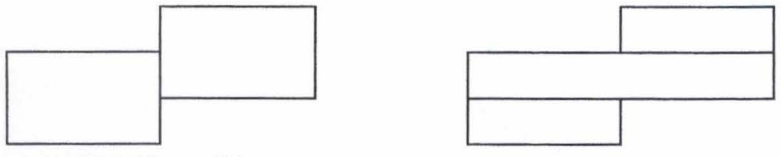


Figure 4.8: Synthèse graphique de deux enregistrements .

Afin de bien comprendre le processus de synthèse de deux enregistrements dénommés respectivement « 1 » et « 2 », celui-ci a été divisé en trois étapes successives qui correspondent chacune au calcul d'un enregistrement ou rectangle résultat.

La première étape calcule le rectangle central « c ». Ce calcul peut être réalisé simplement en prenant pour chaque extrémité respectivement le plus petit des deux `V_DEBUT`, le plus grand des deux `V_FIN`, le plus grand des deux `T_DEBUT` et le plus petit des deux `T_FIN` de « 1 » et de « 2 ». Comme il existe toujours un rectangle central entre deux rectangles contigus latéralement, cette étape existera toujours.

Contrairement à la première étape, la seconde étape n'est pas obligatoire. L'existence de cette étape dépend du taux de recouvrement du rectangle central « c » sur le rectangle « 1 ». En effet, si celui-ci recouvre totalement le rectangle « 1 », la seconde étape n'est pas nécessaire. Ceci est facilement calculable à l'aide des hauteurs des deux rectangles. S'il existe une différence, cela signifie que le rectangle central ne suffit pas pour représenter l'ensemble des informations véhiculées par le rectangle « 1 ». Le rectangle « a » ou les rectangles « a » et « a' » doivent donc être calculés. Grâce au calcul de la différence nous pouvons savoir avec exactitude, afin d'agir adéquatement, dans quel cas de figure nous nous trouvons exactement (figure 4.9).

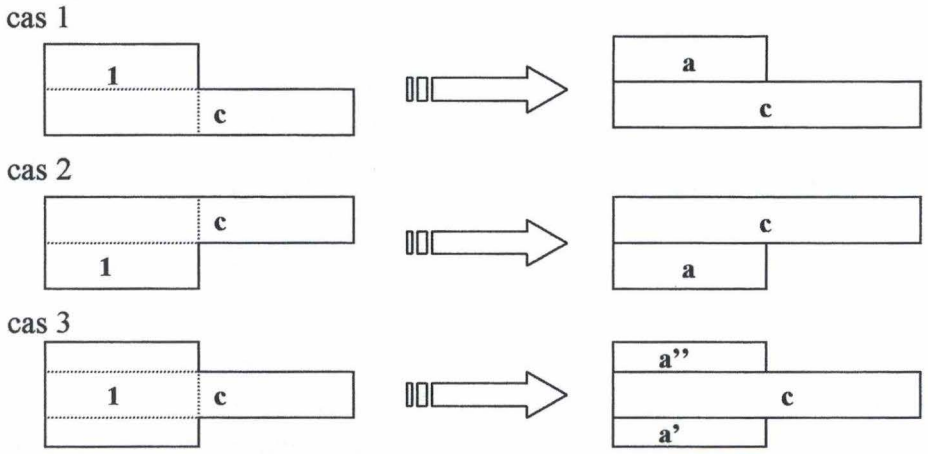


Figure 4.9: Résultat de la synthèse graphique selon les différentes positions du rectangle central « c » par rapport au rectangle « 1 ».

La troisième étape est l'image exacte de l'étape précédente pour le rectangle « 2 ». Les différentes positions des deux rectangles et leurs synthèses sont illustrés à la figure 4.10.

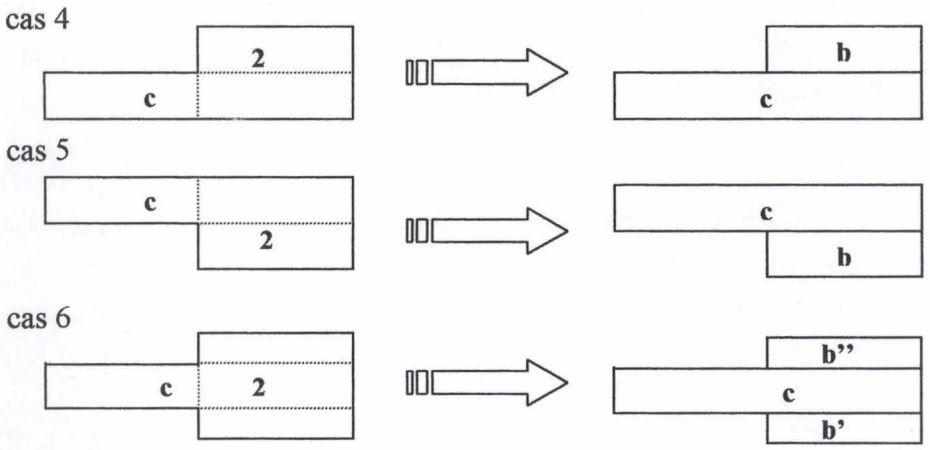


Figure 4.10: Résultat de la synthèse graphique selon les différentes positions du rectangle central « c » par rapport au rectangle « 2 ».

Finalement, les rectangles « a » ou « a » et « a' », « b » ou « b » et « b' » et « c » forment le résultat de la synthèse des deux enregistrements « 1 » et « 2 ».

Le problème de la tenaille est un problème rare qui peut fausser le résultat de notre synthèse. Ce problème est illustré par la figure 4.11 où les numéros expriment l'ordre dans lequel les enregistrements sont traités.

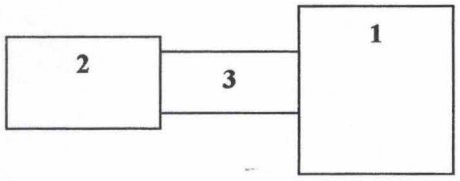


Figure 4.11 : Illustration du problème de la tenaille.

Supposons que les trois enregistrements véhiculent la même information. Dans ce cas, lorsque nous traitons le 3ème enregistrement, notre table de travail possède déjà en mémoire les enregistrements 1 et 2. Ceux-ci ne se sont cependant pas synthétisés ensemble puisqu'ils ne sont pas jointifs latéralement. Lorsque nous traitons le 3ème enregistrement, celui-ci se synthétise avec le premier rectangle qui lui est contigu, soit 1, soit 2, laissant de côté le rectangle restant. C'est pour cette raison que nous devons repasser dans la table de travail pour vérifier si toutes les synthèses ont bien eu lieu.

La **seconde phase** consiste à éliminer les problèmes verticaux. Ce type de problème apparaît lorsque deux rectangles ou enregistrements véhiculant la même information partagent totalement un de leurs côtés supérieurs ou inférieurs. Dans ce cas, nous devons synthétiser les deux rectangles. La figure 4.12 nous montre la réalisation de cette synthèse.

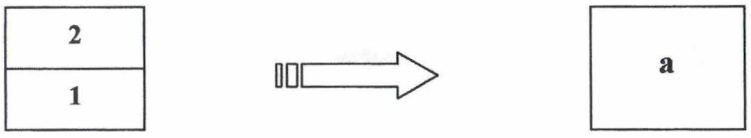


Figure 4.12: Synthèse de deux enregistrements en vue d'éliminer un problème vertical.

La solution est simple à réaliser. Premièrement, la table résultant de la première phase doit être ordonnée selon l'identifiant d'entité, V_DEBUT et T_DEBUT. Ensuite, il suffit de considérer notre table bitemporelle comme une table mono-temporelle avec « transaction time » et de réaliser un simple coalescing. Cette solution est exposée à la figure 4.13 où les tables PERSONNE_5 et PERSONNE_6 représentent respectivement la table avant et après ce coalescing.

PERSONNE_5					
NOM	**ADR***	**V_DEB**	**V_FIN**	**T_DEB**	**T_FIN**
...	...	...	...	...	...
Léo	Namur	10	30	40	45
Léo	Namur	10	30	45	55
...	...	...	...	...	...
Léo	Bruxelles	30	999	40	50
Léo	Bruxelles	30	999	50	60
...	...	...	...	...	...

PERSONNE_6					
NOM	**ADR***	**V_DEB**	**V_FIN**	**T_DEB**	**T_FIN**
...	...	...	...	...	...
Léo	Namur	10	30	40	55
...	...	...	...	...	...
Léo	Bruxelles	30	999	40	60
...	...	...	...	...	...

Figure 4.13: Exemple de résolution des problèmes verticaux.

Un exemple complet de l'application de l'algorithme de coalescing à deux phases sur la table PERSONNE_4 se trouve à l'annexe A.

Cet algorithme à deux phases comporte un inconvénient majeur. En effet, nous savons que les bases de données temporelles détiennent un nombre important d'enregistrements et donc, qu'elles sont gourmandes en place mémoire. Toutefois, lorsque nous voulons réaliser le coalescing sur une base de données bitemporelle, nous devons prévoir un espace mémoire qui peut aller jusqu'au triple de l'espace mémoire requis pour la table de départ. En effet, lors de la première phase, une première table résultat est créée. Celle-ci voit son nombre d'enregistrements augmenter ou diminuer en fonction de la survenance de cas de synthèse particuliers. En fait, le nombre d'enregistrements pourrait augmenter lorsque nous procédons aux synthèses de type 3 ou 6. L'espace mémoire occupé par cette table résultat sera donc à peu près le même que celui occupé par la table de départ. Lors de la seconde phase, une nouvelle table résultat est générée. Celle-ci possède un nombre d'enregistrements équivalent à celui de la table résultat précédente excepté les enregistrements éliminés pour résoudre les problèmes verticaux. Bien que la table résultat de la première phase puisse être effacée après la réalisation de la seconde phase, à un moment donné, nous sommes obligés d'avoir de la place mémoire pour ces 3 tables différentes.

L'algorithme de coalescing à deux phases ne comporte pas que des désavantages. Celui-ci permet d'obtenir de manière rapide le résultat du coalescing en ne parcourant qu'une seule fois la table de départ et la table résultant de la première phase. Durant cette phase, nous parcourons pour

chaque enregistrement deux fois la table de travail (problème de la tenaille). Bien que la table de travail soit en mémoire centrale, notre calcul d'efficacité doit en tenir compte. Pour pouvoir évaluer l'impact sur le temps de cette double lecture, il faut que nous puissions déterminer le nombre d'enregistrements qui y sont stockés. De plus, cette étude nous permettra d'évaluer la capacité de mémoire centrale utile pour réaliser le coalescing. Après calculs, la longueur maximale de la table de travail a pu être déterminée. Celle-ci équivaut au nombre maximum d'enregistrements possibles à un temps t déterminé (t appartenant au « transaction time ») plus un.

Grâce à certaines améliorations, nous pouvons encore augmenter la vitesse d'exécution du coalescing et diminuer l'espace disque requis. Pour ce faire, nous allons réunir les deux phases de l'algorithme précédent en une seule, en vue d'éliminer une des deux lectures-disque nécessaires. L'inconvénient de cette amélioration réside dans la création d'une nouvelle table de travail qui consommera de l'espace mémoire. Celle-ci permettra de stocker les enregistrements qui ne seront plus confrontés aux problèmes horizontaux mais qui sont encore susceptibles de rencontrer des problèmes verticaux.

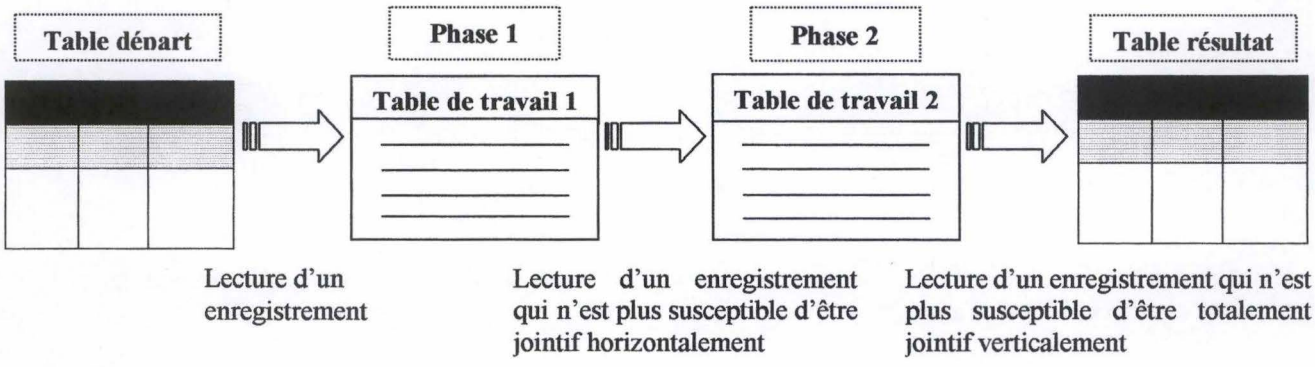


Figure 4.14: Parcours d'un enregistrement dans l'algorithme de coalescing en une phase.

Pour diminuer la taille de nos tables de travail, nous devons nous assurer qu'à chaque fois qu'un enregistrement n'appartient plus à la première table de travail, il est traité directement pour un éventuel problème vertical dans la seconde. En effet, si nous agissons de la sorte, la taille de la seconde table de travail diminuera au fur et à mesure que les enregistrements à stocker dans la table résultat sont déterminés. Un enregistrement de la seconde table de travail est considéré comme un enregistrement final lorsque nous sommes sûrs qu'il ne sera plus totalement jointif verticalement à un autre enregistrement. Cette condition sera vérifiée pour un enregistrement donné lorsqu'un autre enregistrement lui sera jointif supérieurement mais pas totalement. Cette condition est valable grâce aux hypothèses de continuité du « valid time » et du « transaction time ». En effet, sous ces hypothèses, nous pouvons affirmer que tout enregistrement non valide possède au moins un enregistrement qu'il lui est graphiquement jointif supérieurement.

L'algorithme en une seule phase est donné à la figure 4.15. Celui-ci doit être appliqué à chaque entité de la table bitemporelle à synthétiser. Un exemple complet de l'application de l'algorithme de coalescing à une phase sur la table `PERSONNE_4` se trouve à l'annexe B.

```
Tant qu'il existe un enregistrement C1 de l'entité courante E à traiter faire :
{
 Si la table de travail T1 est vide
 Alors C1 est inséré directement dans T1 vide
 Sinon  /*C1 est inséré dans T1 qui est non vide*/
   Tant qu'il existe un enregistrement C1' dans T1 à traiter et
         que C1 n'est pas encore synthétisé faire
    {
     Si C1 possède exactement les mêmes valeurs d'attributs que C1' et
        si C1 et C1' sont latéralement contigus
     Alors C1 et C1' sont synthétisés
     Passer à l'enregistrement C1' de T1 suivant.
    }
  Si  C1 n'a pas été synthétisé
  Alors C1 est inséré dans T1
  /*il se peut que certains éléments n'aient pas été synthétisés*/
  Sinon T1 doit être réexaminé pour vérifier que toutes les synthèses ont bien eu lieu
  Tant qu'il existe un enregistrement C2 de T1 à retirer faire
   {
    Si la table de travail T2 est vide
    Alors C2 est inséré directement dans T2 vide
    Sinon /*C2 est inséré dans T2 qui est non vide*/
    Tant qu'il existe un enregistrement C2' dans T2 à traiter et
          que C2 n'est pas encore synthétisé faire
     {
      Si C2 possède exactement les mêmes valeurs d'attributs que C2' et
         si C2 et C2' sont totalement jointifs verticalement.
      Alors C2 et C2' sont synthétisés
      Passer à l'enregistrement C2' de T2 suivant
     }
     Si C2 n'a pas été synthétisé
     Alors C2 est inséré dans T2
     Tous les enregistrements de T2 qui ne sont plus susceptibles d'être totalement jointif
      verticalement sont insérés dans la table résultat R.
     Passer à l'enregistrement C2 suivant
   }
Passer à l'enregistrement C1 suivant
}
Si T1 ou T2 n'est pas vide
   Alors vider T1 et T2 en suivant le même algorithme
```

Figure 4.15: Algorithme de coalescing en une seule phase.

Le nombre d'enregistrements stockés dans nos tables de travail peut encore diminuer en ordonnant nos enregistrements de départ d'une autre manière. Si nous trions nos enregistrements par leur contenu puis par T_DEBUT et V_DEBUT, nous aurions pour la table PERSONNE le résultat de la figure 4.16.

PERSONNE_7					
NOM	**ADRESSE***	**V_DEBUT**	**V_FIN**	**T_DEBUT**	**T_FIN**
Léo	Bruxelles	30	999	40	50
Léo	Bruxelles	30	999	50	60
Léo	Bruxelles	35	999	60	999
Léo	Liège	20	25	30	40
Léo	Liège	25	999	30	40
Léo	Liège	10	30	55	60
Léo	Liège	10	35	60	999
...	...	...	...	...	...

Figure 4.16: Exemple d'enregistrements de la table PERSONNE_7 triés par leur contenu, T_DEBUT et V_DEBUT.

L'avantage est double car non seulement, nous réduisons la rotation des tables de travail mais nous favorisons aussi le passage plus rapide d'un enregistrement d'une table de travail à l'autre. En effet, avant l'introduction de cette amélioration, la rotation des tables de travail se faisait après chaque entité. En d'autres termes, nous vidions les tables de travail après chaque entité car nous avions la certitude que tous les enregistrements qui suivraient ne véhiculeraient plus jamais la même information. Maintenant avec l'introduction de cette amélioration, puisque les enregistrements sont déjà classés en fonction de l'information véhiculée, nous pouvons vider nos tables de travail dès que chacune de celles-ci a été traitée. Par exemple, lorsque l'information « Léo habite Bruxelles » a été traitée, les tables de travail peuvent être vidées. Deuxièmement, puisque nous n'ordonnons plus nos enregistrements pour chaque entité en fonction du temps mais en fonction du contenu, nous avons beaucoup plus de chance d'avoir des éléments non jointifs et fort dispersés dans le temps. Ce phénomène accélèrerait les conditions de sortie des enregistrements de chacune des deux tables de travail.

Pour que cette amélioration soit effective, la condition de sortie de la seconde table de travail doit être changée. En effet, maintenant, lorsqu'un enregistrement de la seconde table de travail est inférieur en terme de « transaction time » à l'enregistrement entrant, il sera considéré comme final

4.4 Clé étrangère temporelle

Partons d'un exemple pour expliquer ce qu'est une clé étrangère temporelle. La table PERSONNE de la figure 4.17 enregistre l'évolution dans le temps de l'adresse et du salaire de certaines personnes. L'identifiant primaire de cette table est {NOM, DEBUT}. Considérons maintenant l'existence d'une autre table temporelle : la table VILLE qui enregistre l'évolution au cours du temps du nombre d'habitants (en milliers) de chaque ville. Son identifiant est {LOCALITE,

DEBUT}. Les tables PERSONNE et VILLE, ainsi que toutes les tables que nous étudions, répondent aux hypothèses de travail émises au chapitre 3. Ces tables sont dites normalisées.

PERSONNE				
NOM	**ADRESSE***	**SALAIRE***	**DEBUT**	**FIN**
Léo	Liège	50000	10	35
Léo	Bruxelles	100000	35	999
René	Namur	20000	15	25
René	Namur	30000	25	50
René	Namur	50000	50	75
René	Bruxelles	50000	75	90
Jean	Liége	25000	20	25
Jean	Namur	25000	25	40
Jean	Namur	60000	40	999

VILLE			
LOCALITE	**NBR_HAB***	**DEBUT**	**FIN**
Bruxelles	900	10	80
Bruxelles	1000	80	999
Liège	450	5	30
Liège	500	30	999
Namur	200	5	30
Namur	220	30	65
Namur	250	65	999

VILLE_EXIST		
LOCALITE	**DEBUT**	**FIN**
Bruxelles	10	999
Liège	5	999
Namur	5	65

Figure 4.17: Exemple d'enregistrements des tables mono-temporelles PERSONNE, VILLE et VILLE_EXIST.

D'après les données de ces deux tables, l'attribut ADRESSE de la table PERSONNE possède une clé étrangère temporelle vers la table VILLE. En effet, cette constatation sera confirmée par l'analyse de toutes les conditions d'existence d'une clé étrangère temporelle. Afin de bien montrer la différence existant entre une clé étrangère simple et une clé étrangère temporelle, les conditions d'existence de ces deux types de clé sont reprises ci-après.

- **Clé étrangère simple de l'attribut ADRESSE vers la table VILLE**

  PERSONNE[ADRESSE] $\subseteq$ VILLE[LOCALITE]

- **Clé étrangère temporelle de l'attribut ADRESSE vers la table VILLE**

  Tout intervalle de temps [DEBUT,FIN[de chaque enregistrement, associé à une ville particulière, de la table PERSONNE doit être inclus dans l'intervalle d'existence de cette même ville dans la table VILLE.

La période durant laquelle une entité existe (intervalle d'existence) est déterminée en prenant respectivement la plus petite valeur de début et la plus grande valeur de fin des enregistrements de cette entité . Pour une meilleure compréhension, les intervalles d'existence de chacune des entités de la table VILLE sont représentés par la table VILLE_EXIST de la figure 4.17. Pour plus de précision sur les clés étrangères temporelles, le lecteur est invité à consulter [HAINAUT, 99].

4.5 Jointure temporelle

La jointure temporelle se distingue de la jointure normale par sa condition de sélection plus complexe. Celle-ci prend en compte les intervalles de temps. Par exemple, il n'est pas correct d'affirmer à la figure 4.17 que Jean vivait à Liège avec un salaire de 25000 francs lorsque Liège comportait 500.000 habitants. En effet, celui-ci a bien vécu à Liège entre le 20 et le 25 mais à cette époque Liège ne comportait que 450.000 habitants. La jointure temporelle va donc travailler sur des enregistrements dont les intervalles de temps sont non disjoints.

De plus, lors d'une jointure temporelle de deux tables, la valeur de l'intervalle de temps des enregistrements obtenus doit être recalculée. Celui-ci correspond à la partie commune des deux intervalles de temps considérés. La figure 4.18 illustre ce cas. Si nous considérons l'entité Léo et son passage à Liège, deux nouveaux enregistrements avec de nouvelles valeurs d'intervalles sont générés lors de la jointure temporelle avec la table VILLE. L'illustration graphique de la jointure de ces deux enregistrements se trouve à la figure 4.19.

PERSONNE_VILLE					
NOM	**ADRESSE***	**NBR_HAB***	**SALAIRE***	**DEBUT**	**FIN**
...	...	...	...	...	...
Léo	Liège	450	50000	10	30
Léo	Liège	500	50000	30	35
...	...	...	...	...	...

Figure 4.18: Résultat de la jointure d'un enregistrement de la table PERSONNE avec la table VILLE.

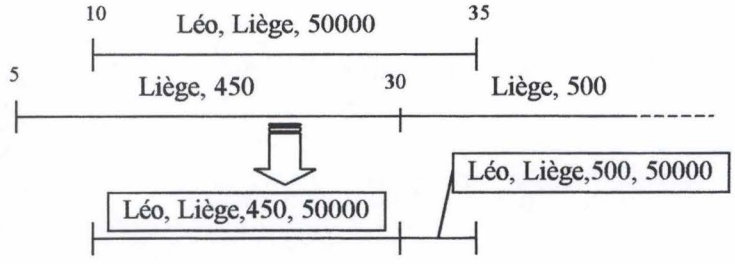


Figure 4.19: Illustration graphique de la jointure d'un enregistrement de la table PERSONNE avec la table VILLE.

La différence existant entre une requête de jointure normale et temporelle est flagrante. Un exemple sur les deux tables PERSONNE et VILLE, nous en persuadera.

Jointure normale :

```
select NOM, ADRESSE, NBR_HAB, SALAIRE
from   PERSONNE , VILLE
where  ADRESSE=LOCALITE
```

Jointure temporelle

```
select NOM,ADRESSE,NBR_HAB,SALAIRE,IMAX(P.DEBUT,V.DEBUT),IMIN(P.FIN,V.FIN)
from   PERSONNE P,VILLE V
where  P.ADRESSE=V.LOCALITE
and    (P.DEBUT < V.FIN) and (V.DEBUT < P.FIN)
```

La jointure temporelle décrite ci-dessus n'est pas correcte car il n'existe pas en SQL92 d'opérateur `IMAX` et `IMIN` qui permettrait respectivement de calculer le maximum et le minimum de deux colonnes. Si nous travaillons avec un SQL étendu, grâce à la fonction *case* le remplacement de ces deux opérateurs est possible [HAINAUT, 99]. [HAINAUT, 99] attire aussi notre attention sur la complexité que prendrait une requête de jointure temporelle si cette fonction *case* n'existait pas. Bien que SQL92 ne possède pas cette fonction, `IMAX` et `IMIN` seront utilisés pour une meilleure lisibilité de nos requêtes.

Pour l'instant, seule la jointure temporelle de deux tables mono-temporelles a été expliquée. Cependant, comme il existe différents types de tables temporelles, la jointure temporelle sera réalisable entre ces différentes tables. La **grille d'analyse** de la figure 4.20 met en évidence les différentes possibilités de jointure.

↱	NT	MV	MT	B
NT	⬭	☼	☼	☼
MV	☼	☼		☼
MT	☼		☼	
B	☼	☼		☼

Légende

- **NT** : table non-temporelle
- **MV** : table mono-temporelle avec le valid time
- **MT** : table mono-temporelle avec le transaction time
- **B** : table bi-temporelle
- ↱ : exprime le lien entre le type de table dont provient la référence temporelle et le type de table référencée.
- ⬭ : jointure normale.
- ☼ : jointure temporelle.

Figure 4.20 : Grille d'analyse reflétant tous les cas de jointures possibles entre deux tables.

Afin d'améliorer notre compréhension de la jointure temporelle, chacune des possibilités de cette grille va être examinée. En effet, pour chacun des cas, la contrainte d'intégrité générée par l'apparition de la clé étrangère normale ou temporelle ainsi que la requête de jointure associée seront exposées. Pour rester concret, l'exemple des deux tables PERSONNE et VILLE sera utilisé (figure 4.17).

Cas 1 : NT ⇒ NT

Le cas 1 est le cas normal d'une table non temporelle possédant une clé étrangère normale vers une autre table non temporelle.

La contrainte d'intégrité normale:

```
PERSONNE[ADRESSE] ⊆ VILLE[LOCALITE]
```

La requête de jointure normale:

```
select NOM,ADRESSE,NBR_HAB,SALAIRE
from   PERSONNE,VILLE
where  ADRESSE=LOCALITE
```

Cas 2 : NT ⇒ MV et Cas 3 : NT ⇒ MT

Ce sont les cas où une table non temporelle possède une clé étrangère vers une table mono-temporelle. Puisqu'une table non temporelle exprime notre connaissance actuelle du mini-monde, la jointure temporelle entre ces deux types de tables doit relater uniquement les informations présentes. En d'autres termes, seuls les états courants de la table mono-temporelle nous intéressent.

La contrainte d'intégrité temporelle:

```
PERSONNE[ADRESSE] ⊆ présent(VILLE[LOCALITE])
```

L'opérateur *présent()* sélectionne les enregistrements courants d'une table temporelle.

La requête de jointure temporelle:

```
select NOM,ADRESSE,NBR_HAB,SALAIRE
from   PERSONNE,VILLE
where  ADRESSE=LOCALITE
and    FIN=999
```

Cas 4 : NT ⇒ B

C'est le cas où une table non temporelle possède une clé étrangère vers une table bitemporelle. Le cas 4 est l'image des cas 2 et 3 précédent pour une table bitemporelle.

La contrainte d'intégrité temporelle:

```
PERSONNE[ADRESSE] ⊆ présent(VILLE[LOCALITE])
```

La requête de jointure temporelle:

```
select NOM,ADRESSE,NBR_HAB,SALAIRE
from   PERSONNE,VILLE
where  ADRESSE=LOCALITE
and    V_FIN=999  and  T_FIN=999
```

Cas 5 : MV ⇒ MV et Cas 6 : MT⇒ MT

Ce sont les cas où une table mono-temporelle possède une clé étrangère temporelle vers une autre table mono-temporelle. C'est ce type de cas qui a servi d'introduction à l'explication de la jointure temporelle.

La contrainte d'intégrité temporelle:

Tout intervalle de temps [DEBUT,FIN[de chaque enregistrement, associé à une ville particulière, de la table PERSONNE doit être inclus dans l'intervalle d'existence de cette même ville dans la table VILLE.

La requête de jointure temporelle:

```
select NOM,ADRESSE,NBR_HAB,SALAIRE,IMAX(P.DEBUT,V.DEBUT),IMIN(P.FIN,V.FIN)
from   PERSONNE P,VILLE V
where  P.ADRESSE=V.LOCALITE
and    (P.DEBUT < V.FIN) and (V.DEBUT < P.FIN)
```

Cas 7 : MV ⇒ B

C'est le cas où une table mono-temporelle avec « valid time » possède une clé étrangère temporelle vers une table bitemporelle. Puisqu'une table mono-temporelle exprime notre connaissance **actuelle** sur le passé, le présent et le futur, la jointure entre ces deux types de tables doit relater uniquement ces informations. En d'autres termes, lors de la jointure, nous ne nous intéresserons pas aux états non valides de la table bitemporelles.

La contrainte d'intégrité temporelle:

Tout intervalle de temps [V_DEBUT,V_FIN[de chaque enregistrement, associé à une ville particulière, de la table PERSONNE doit être inclus dans l'historique courant de cette même ville dans la table VILLE.

La requête de jointure temporelle:

```
select NOM,ADRESSE,NBR_HAB,SALAIRE,IMAX(P.V_DEBUT,V.V_DEBUT),IMIN(P.V_FIN,V.V_FIN)
from   PERSONNE P,VILLE V
where  P.ADRESSE=V.LOCALITE
and    (P.V_DEBUT < V.V_FIN) and (V.V_DEBUT < P.V_FIN)
and    V.T_FIN=999
```

Cas 8 : B ⇒ B

C'est le cas où une table bitemporelle possède une clé étrangère temporelle vers une autre table bitemporelle. Le cas 8 est similaire aux cas 5 et 6 à la différence près que nous travaillons avec une dimension en plus. Graphiquement, à la place des segments de droite, nous allons rechercher les rectangles non disjoints. La figure 4.21 met en évidence l'apparition de cette nouvelle dimension. Considérons l'enregistrement « Léo habitait à Bruxelles avec un salaire de 30.000 francs ». L'entité référencée est celle de Bruxelles. Le résultat de la jointure temporelle (figure 4.21) prend donc en compte toutes les informations historiques de nos connaissances sur l'évolution du nombre d'habitant de la ville de Bruxelles.

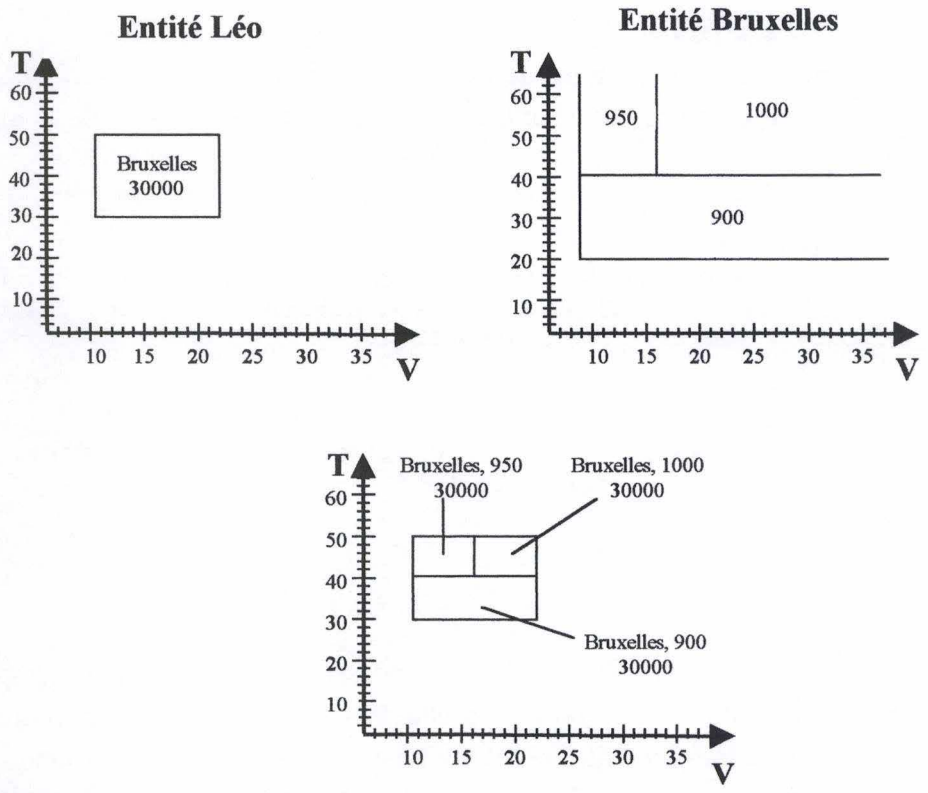


Figure 4.21: Jointure temporelle d'un enregistrement entre deux tables bitemporelles.

La contrainte d'intégrité temporelle:

```
Tout rectangle de chaque enregistrement, associé à une ville particulière, de
la table PERSONNE doit être inclus dans le rectangle d'existence de cette même
ville dans la table VILLE.
```

La requête de jointure temporelle:

```
select NOM,ADRESSE,NBR_HAB,SALAIRE,IMAX(P.V_DEBUT,V.V_DEBUT),IMIN(P.V_FIN,V.V_FIN),
       IMAX(P.T_DEBUT,V.T_DEBUT),IMIN(P.T_FIN,V.T_FIN)
from   PERSONNE P,VILLE V
where  P.ADRESSE=V.LOCALITE
and    (P.V_DEBUT < V.V_FIN) and (V.V_DEBUT < P.V_FIN)
and    (P.T_DEBUT < V.T_FIN) and (V.T_DEBUT < P.T_FIN)
```

Les cas qui suivent sont particuliers. En effet, comme la table référencée comporte moins d'informations, en termes temporels, que la table dont provient la référence, certains enregistrements ne sont joints à aucune valeur. Afin de bien expliquer ces cas spéciaux, nous détaillerons le plus compliqué d'entre eux (B $\Rightarrow$ MV).

Cas 9 : B $\Rightarrow$ MV

Dans le cas 9, les enregistrements non valides de la table bitemporelle ne sont joints à aucune des valeurs de la table mono-temporelle. Mais, nous pourrions en vue d'obtenir le résultat utiliser la valeur `null` pour combler le manque d'information temporelle de la table MV. La requête de jointure temporelle pourrait être écrite grâce à l'utilisation d'une `union`.

```
select NOM,ADRESSE,NBR_HAB,SALAIRE,IMAX(P.V_DEBUT,V.V_DEBUT),IMIN(P.V_FIN,V.V_FIN)
       P.T_DEBUT,P.T_FIN
from   PERSONNE P, VILLE V
where  P.ADRESSE = V.LOCALITE
and    (P.V_DEBUT < V.V_FIN) and (V.V_DEBUT < P.V_FIN)
and    P.T_FIN = 999
union
select NOM,ADRESSE,null,SALAIRE,V_DEBUT,V_FIN,T_DEBUT,T_FIN
from   PERSONNE
where  T_FIN <> 999
```

Mais même dans ce cas de figure et en procédant à la requête de jointure temporelle ci-dessus, le résultat obtenu ne reflètera pas exactement la signification temporelle des deux bases de données. L'exemple de la figure 4.22 nous le prouve. En effet, nous ne pouvons pas affirmer, par exemple, avec pour seules connaissances les tables PERSONNE et VILLE, que nous savions à partir du 20 jusqu'à maintenant que Léo a vécu à Namur avec un salaire de 50.000 francs entre le 10 et le 30 lorsque Namur comptait 200.000 habitants. Il est vrai qu'actuellement nous connaissons l'évolution du nombre d'habitants de la ville de Namur mais, en aucun cas, nous ne pouvons affirmer que cette connaissance de l'évolution est restée constante durant la période du 20 jusqu'à maintenant. En effet, seule la connaissance actuelle est représentée dans une base de données mono-temporelle. La solution exacte consiste à clôturer nos connaissances passées et à mettre à jour nos connaissances présentes (figure 4.22).

PERSONNE						
NOM	**ADRESSE***	**SALAIRE***	**V_DEBUT**	**V_FIN**	**T_DEBUT**	**T_FIN**
Léo	Namur	50000	10	40	20	999
...	...	...	...	...	...	...

VILLE			
LOCALITE	**NBR_HAB***	**DEBUT**	**FIN**
Namur	200	5	30
Namur	220	30	65
Namur	250	65	999
....	...	...	...

INCORRECT

PERSONNE_VILLE							
NOM	**ADRESSE***	**NBR_HAB***	**SALAIRE***	**V_DEBUT**	**V_FIN**	**T_DEBUT**	**T_FIN**
Léo	Namur	200	50000	10	30	20	999
Léo	Namur	220	50000	30	40	20	999
...	...	...	...	...	...	...	...

CORRECT

PERSONNE_VILLE							
NOM	**ADRESSE***	**NBR_HAB***	**SALAIRE***	**V_DEBUT**	**V_FIN**	**T_DEBUT**	**T_FIN**
Léo	Namur	NULL	50000	10	40	20	70
Léo	Namur	200	50000	10	30	70	999
Léo	Namur	220	50000	30	40	70	999
...	...	...	...	...	...	...	...

Figure 4.22: Exemple de jointure temporelle entre une table bitemporelle et une table mono-temporelle (B⇒MV) à la date du 70.

La contrainte d'intégrité temporelle :

Toute période de validité de chaque enregistrement dont la période de transaction contient le moment présent, associé à une ville particulière, de la table PERSONNE doit être incluse dans l'intervalle d'existence de cette même ville dans la table VILLE.

La requête de jointure temporelle correcte:

```
select NOM,ADRESSE,NBR_HAB,SALAIRE,IMAX(P.V_DEBUT,V.V_DEBUT),IMIN(P.V_FIN,V.V_FIN),
       maintenant,999
from   PERSONNE P,VILLE V
where  P.ADRESSE = V.LOCALITE
and    (P.V_DEBUT < V.V_FIN) and (V.V_DEBUT < P.V_FIN)
and    P.T_FIN = 999
union
select NOM,ADRESSE,null,SALAIRE,V_DEBUT,V_FIN,T_DEBUT,maintenant
from   PERSONNE
where  T_FIN = 999
union
select NOM,ADRESSE,null,SALAIRE,V_DEBUT,V_FIN,T_DEBUT,T_FIN
from   PERSONNE
where  T_FIN <> 999
```

Cas 10 : MV ⇒ NT et Cas 11 : MT⇒ NT

Le cas 10 et le cas 11 sont les cas où une table mono-temporelle possède une clé étrangère normale vers une table non temporelle.

La contrainte d'intégrité:

présent(PERSONNE[ADRESSE]) ⊆ VILLE[LOCALITE]

La requête de jointure normale:

```
select NOM,ADRESSE,NBR_HAB,SALAIRE, maintenant, 999
from   PERSONNE P, VILLE V
where  P.ADRESSE = V.LOCALITE
and    P.FIN = 999
union
select NOM,ADRESSE,null,SALAIRE,DEBUT,maintenant
from   PERSONNE
where  FIN = 999
union
select NOM,ADRESSE,null,SALAIRE,DEBUT,FIN
from   PERSONNE
where  FIN <> 999
```

Cas 12 : B ⇒ NT

C'est le cas où une table bitemporelle possède une clé étrangère normale vers une table non temporelle.

La contrainte d'intégrité temporelle :

```
présent(PERSONNE[ADRESSE]) ⊆ VILLE[LOCALITE]
```

La requête de jointure normale:

```
select NOM,ADRESSE,NBR_HAB,SALAIRE,BUT,maintenant,999, maintenant,999
from   PERSONNE P, VILLE V
where  P.ADRESSE = V.LOCALITE
and    P.V_FIN = 999
and    P.T_FIN = 999
union
select NOM,ADRESSE,null,SALAIRE,V_DEBUT,maintenant,T_DEBUT,maintenant
from   PERSONNE
where  V_FIN = 999
and    T_FIN = 999
union
select NOM,ADRESSE,null,SALAIRE,V_DEBUT,V_FIN,T_DEBUT,T_FIN
from   PERSONNE
where  V_FIN <> 999
or     T_FIN <> 999
```

Pour compléter notre explication détaillée de la jointure temporelle, nous nous devons d'expliquer pourquoi certains cas sont impossibles. Comme le « transaction time » se trouvant dans une base de données mono_temporelle n'a pas exactement la même signification que celui qui se trouve dans une base de données bitemporelle, les cas MT⇒B et B⇒MT ne sont pas envisageables. De même, les cas MT⇒MV et MV⇒MT n'existent pas. En effet, par définition, le « valid time » n'a pas la même signification que le « transaction time ». Les deux concepts ne peuvent donc pas être mélangés.

5
Optimisation

5.1 Introduction

Nous savons que de nos jours le temps d'accès à l'information est crucial. Il en est de même dans le domaine des bases de données temporelles. Le problème auquel nous devons faire face est celui de l'augmentation accrue des enregistrements. Cette augmentation a pour conséquence de diminuer les performances d'accès. En effet, il nous suffit de faire une requête avec une jointure de deux tables de 10.000 enregistrements pour s'en persuader. Il nous faut donc pallier ce problème et trouver un moyen d'optimiser l'accès aux informations des bases de données temporelles (paragraphe 5.2).

5.2 Technique d'optimisation

La solution retenue consiste, lors de la conception d'une base de données temporelle, à passer par une phase de pré-optimisation. Celle-ci permettra au concepteur de développer sa propre politique en terme d'efficacité d'accès et de place mémoire. Concrètement, cette phase de pré-optimisation aura pour but de structurer les données temporelles. Plus précisément, [DETIENNE, 98a] nous propose de structurer les données temporelles selon deux axes repris ci-après.

Selon l'axe des informations temporelles et non temporelles

- Regroupement des colonnes temporelles et non temporelles dans la même table.
- Dissociation des colonnes temporelles et non temporelles dans des tables différentes avec un regroupement de toutes les colonnes temporelles dans une et une seule table.
- Dissociation des colonnes temporelles et non temporelles dans des tables différentes avec une séparation des colonnes temporelles dans différentes tables.

Selon l'axe des états courants et historiques

- Regroupement des états courants et historiques.
- Regroupement des états courants et historiques avec une copie des états courants dans une table spécifique.
- Dissociation des états courants et historiques.

Lors de la pré-optimisation d'une table temporelle, le concepteur doit déterminer la méthode qu'il appliquera selon chacun de ces deux axes. Il existe donc neuf types de structurations ou transformations des données temporelles. [DETIENNE, 98a] énumère les avantages de chacune de ces transformations.

Pour une raison de standardisation, nous allons adopter une dénomination particulière pour les tables issues de la pré-optimisation. Les noms des tables enregistrant les états courants, les états futurs, les états historiques (passés valides) ou les états non valides seront préfixés respectivement par les symboles « C », « F », « H » et « NV ». Le nom de la table pré-optimisée dépendra donc de son type. Par exemple, lorsque nous travaillerons sur une table mono-temporelle, nous ne rencontrerons jamais d'états non valides.

D'après [DETIENNE, 98c], seules trois transformations sont intéressantes. Afin de les détailler, nous utiliserons, dans tous nos exemples, comme colonnes temporelles, les colonnes ADRESSE et SALAIRE.

La **transformation numéro 1** regroupe les colonnes temporelles et non temporelles ainsi que tous les états (C, F, H et NV) dans une seule et même table. Cette transformation sera appliquée par défaut si le concepteur de la base de données ne procède à aucune transformation. Cette transformation ainsi que les suivantes peuvent s'appliquer sur les trois types de tables connues

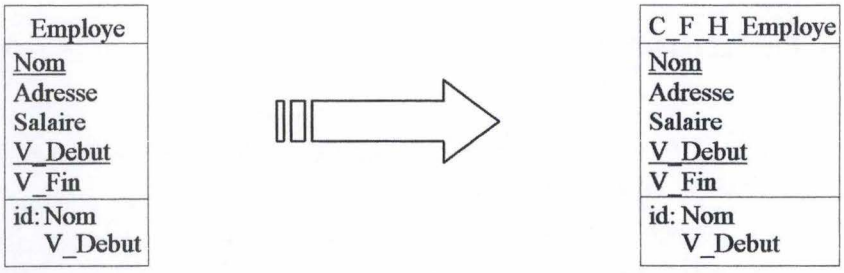


Figure 5.1: Pré-optimisation de la table mono-temporelle avec « valid time » Employe selon la transformation 1.

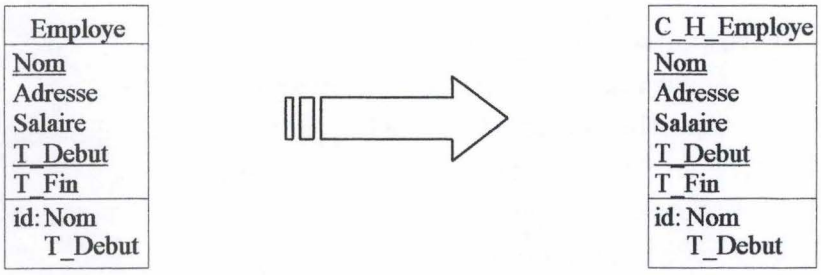


Figure 5.2: Pré-optimisation de la table mono-temporelle avec « transaction time » Employe selon la transformation 1.

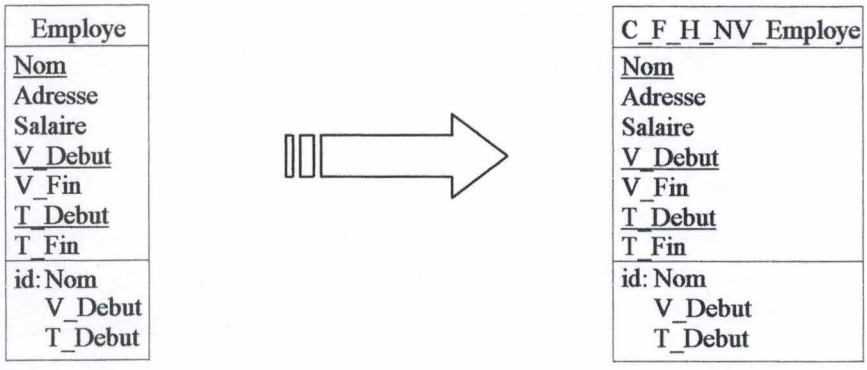


Figure 5.3: Pré-optimisation de la table bitemporelle Employe selon la transformation 1.

La **transformation numéro 2** regroupe les colonnes temporelles et non temporelles mais dissocie les états courants (C) des états passés et non valides (H et NV).

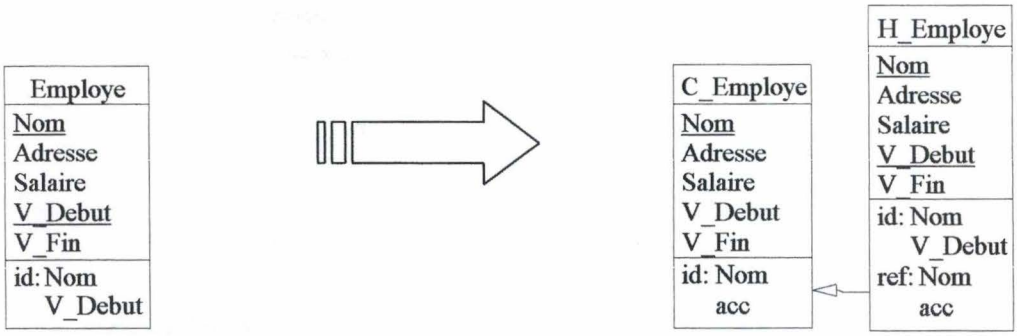


Figure 5.4: Pré-optimisation de la table mono-temporelle avec « valid time » Employe selon la transformation 2.

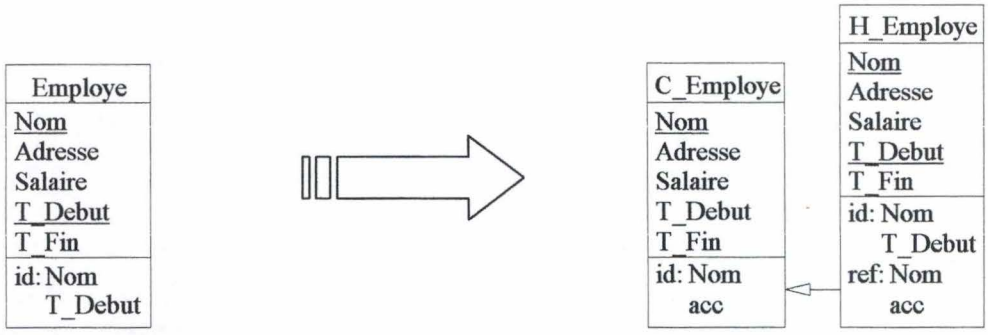


Figure 5.5: Pré-optimisation de la table mono-temporelle avec « transaction time » Employe selon la transformation 2.

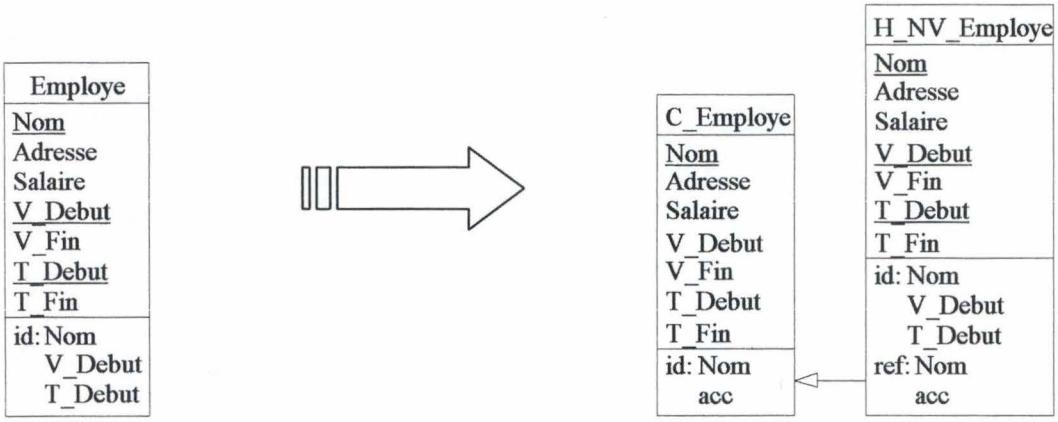


Figure 5.6: Pré-optimisation de la table bitemporelle Employe selon la transformation 2.

La **transformation numéro 3** regroupe les colonnes temporelles et non temporelles et tous les états dans une même table (C, H et NV) tout en créant une copie des états courants (C).

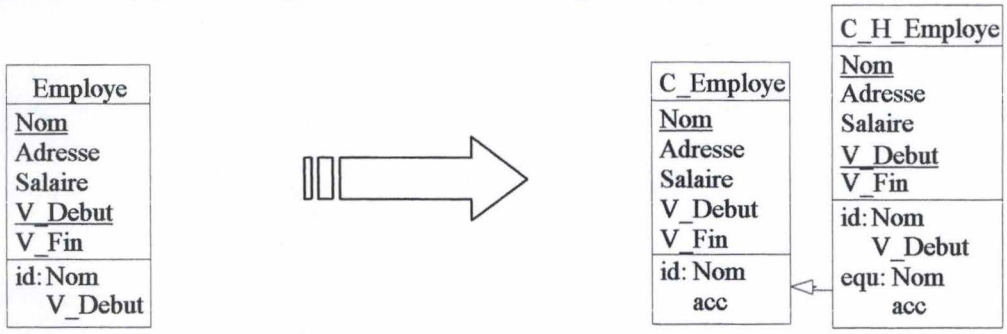


Figure 5.7: Pré-optimisation de la table mono-temporelle avec « valid time » Employe selon la transformation 3.

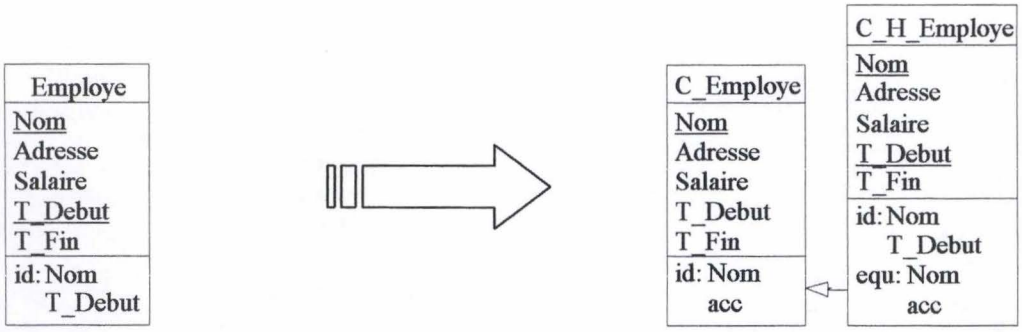


Figure 5.8: Pré-optimisation de la table mono-temporelle avec « transaction time » Employe selon la transformation 3.

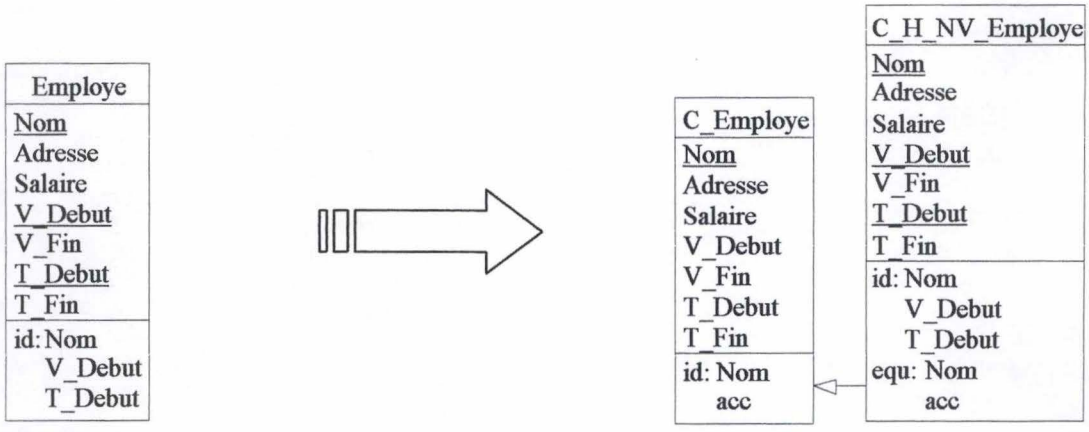


Figure 5.9: Pré-optimisation de la table bitemporelle Employe selon la transformation 3.

Nous pouvons remarquer que seules les bases de données qui ont subi la transformation 1, excepté les bases de données mono-temporelles avec « transaction time », peuvent enregistrer des évènements futurs. En effet, [DETIENNE, 98c] souligne la difficulté de gérer les données futures dans les deux autres transformations car celles-ci nécessiteraient un contrôle constant des enregistrements pour déterminer ceux qui sont courants.

6
Un langage de requêtes temporelles : le langage mini-TSQL2

6.1 Introduction

Il existe une multitude de langages de requêtes qui ont été proposés en vue de pouvoir accéder aux données des bases de données temporelles. C'est pour cette raison que ce chapitre va, tout d'abord, dégager les caractéristiques fondamentales de ces langages temporels (paragraphe 6.2). Ensuite, l'étude de ces caractéristiques nous permettra d'élaborer à notre tour et de manière efficiente, pour nos propres bases de données, un langage temporel (paragraphe 6.3) dont nous donnerons la syntaxe et la sémantique complète (paragraphe 6.4 et 6.5). Nous illustrerons ceci à l'aide de quelques exemples (paragraphes 6.6 et 6.7). Afin de bien comprendre la portée de notre langage, nous nous interrogerons aussi sur la puissance de celui-ci (paragraphe 6.8).

6.2 Caractéristiques d'un bon langage temporel

Une des plus importantes caractéristiques d'un langage temporel est de donner la possibilité à l'utilisateur d'extraire les états courants, historiques et futurs d'une base de données temporelle. Nous pouvons regrouper les critères d'un bon langage temporel en deux catégories : les critères de capacité et les critères de design. Nous trouvons dans le premier groupe ce que les langages devraient fournir pour être de bons langages temporels. Quant au second groupe, il met en évidence les caractéristiques nécessaires pour rester proche des connaissances de l'utilisateur [KARVELIS, 94]. Malgré qu'il soit souvent négligé par les concepteurs, ce deuxième groupe est la clé de succès de l'adoption par les utilisateurs d'un langage temporel.

Les critères de capacité :

- La possibilité de manipuler le « valid time », le « transaction time ».
- La possibilité de stocker des évènements passés, présents et futurs.
- La possibilité de récupérer des évènements passés, présents et futurs.

Les critères de design :

- Un langage simple.
- La syntaxe du langage temporel doit être une extension intuitive d'un langage existant.
- Le langage temporel doit avoir sa sémantique par défaut cohérente avec le langage dont il dérive.

Comme nous travaillons sur des bases de données relationnelles, seuls les langages temporels relationnels nous intéressent. La majorité d'entre eux sont des extensions du langage SQL qui reste le langage le plus répandu dans le monde des bases de données. [KARVELIS, 94] nous donne une courte description de certains langages temporels : **SQL+T**, **SQL^T**, ***Temp*SQL**. Notre langage va principalement s'inspirer du langage **TSQL2** (*Temporal Structured Query Langage*) [SNODGRASS, 95]. Celui-ci a été élaboré par un comité de développement qui s'est réuni sous l'impulsion de Richard T. Snodgrass pour mettre sur pied une spécification de langage temporel qui serait une extension d'SQL-92. En effet, comme la recherche dans le domaine des bases de données temporelles est très fragmentée, Snodgrass a voulu générer une base de travail commune pour les recherches futures. Les avantages de TSQL2 sont nombreux. Il permet entre autres de manipuler des périodes de temps et de supporter des granularités et des calendriers différents. [SNODGRASS, 95] décrit de manière complète l'ensemble des recherches menées sur le langage TSQL2. Cependant, aucune implémentation de TSQL2 n'a encore vu le jour. Les spécifications de TSQL2 sont donc un point de départ intéressant pour l'implémentation de notre propre langage temporel.

6.3 Le langage mini-TSQL2

6.3.1 Généralités

Tout langage de requête peut être découpé en deux grandes parties : le langage de définition des données et de manipulation des données. Nous trouvons respectivement les requêtes de création et de suppression de table dans le premier groupe et les requêtes de sélection, d'insertion, de suppression et de mise à jour des données dans le second. Comme le domaine des bases de données temporelles est très vaste, notre travail se limitera à l'étude d'un langage de manipulation des données spécifiquement adapté à nos types de bases de données temporelles. De plus, nous concentrerons nos efforts uniquement sur les requêtes de sélection. Comme notre langage se base sur les spécifications de TSQL2, celui-ci sera nommé mini-TSQL2.

Avant de nous lancer dans la description syntaxique et sémantique de ce nouveau langage temporel, nous devons essayer de dégager exactement les souhaits des utilisateurs dans le domaine de la sélection de données temporelles. On peut estimer que leurs principaux desiderata sont au nombre de quatre :

- Sélectionner les états courants de la base de données,
- Sélectionner les connaissances ou l'état de la base de données à un instant T déterminé,
- Sélectionner l'historique des connaissances ou de l'état de la base de données durant une période déterminée,
- Sélectionner l'historique partiel des connaissances ou de l'état de la base de données.

Comme pour le langage SQL dont notre langage dérive, nous allons développer séparément les trois parties principales d'une requête de sélection. En effet, une requête simple de sélection contient une clause `select`, une clause `from` et une clause `where`. En fait, comme pour l'interprétation d'une requête, la clause `from` sera examinée avant la clause `where` et la clause

`select`. De plus, pour travailler de manière progressive, nous traiterons de l'extraction ou de la sélection des données d'une seule table avant d'aborder le cas de deux tables.

6.3.2 Sélection dans une seule table

La clause `from` du langage mini-TSQL2

A première vue, la clause `from` du langage SQL et de notre langage mini-TSQL2 est identique. Toutefois, une différence majeure les sépare. Lorsque nous soumettons au SGBD, à travers la clause `from` d'une requête SQL, le nom de la table à laquelle nous voulons appliquer la requête, cette table existe physiquement dans la base de données, ce qui n'est pas le cas pour une table temporelle avec notre langage mini-TSQL2. En effet, pour des raisons d'optimisation (chapitre 5), lors de la phase de conception, un éclatement de la table temporelle peut se produire. Comme l'utilisateur n'est pas toujours le concepteur de la table, nous ne pouvons pas obliger l'utilisateur à connaître le type de transformation qu'a subi la table temporelle lors de la phase de pré-optimisation. C'est donc pour cette raison que les noms des tables temporelles de la clause `from` de notre langage sont des noms logiques. La figure 6.1 illustre la différence existant entre le monde logique dans lequel le programmeur ou l'utilisateur de notre langage évolue et le monde physique ou réel tel que nous le trouvons dans nos bases de données.

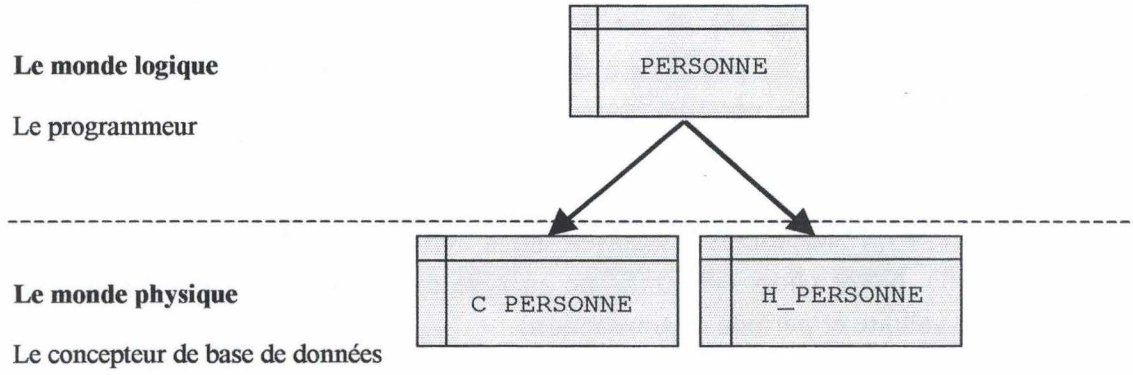


Figure 6.1: Illustration de la différence entre le monde logique et la monde physique.

La clause `where` du langage mini-TSQL2

Pour répondre aux souhaits des programmeurs, les conditions de sélection dans la clause `where` doivent être améliorées en vue de permettre aux programmeurs d'accéder aux données temporelles désirées. C'est pour cette raison que la clause `where` de notre langage diffère de celle du langage SQL essentiellement par l'extension des commandes existantes et l'apparition de nouveaux opérateurs. La clause `where` de notre langage sera composé de deux parties distinctes séparées par une virgule : une pour les conditions temporelles et une autre pour les conditions non temporelles. Comme tous les ouvrages sur SQL relatent en détail la syntaxe et la sémantique des conditions non temporelles, nous allons nous concentrer uniquement sur les conditions temporelles.

Pour réaliser nos conditions temporelles, deux nouveaux types de données doivent être créés : le type TIMEPOINT et le type PERIOD.

Snodgrass utilise le terme « **datetime** » pour exprimer un point isolé dans le temps. Ce point possède une certaine granularité, par exemple, une seconde, une journée ou une année [SNODGRASS, 95]. Dans TSQL2, DATE, TIME ou TIMESTAMP sont les littéraux de granularités différentes utilisés pour déclarer un « datetime » dans le temps. Comme nous l'avons souligné dans l'introduction, pour simplifier notre exposé, nous travaillons dans un modèle doté d'une seule granularité. C'est ainsi qu'un point dans le temps est exprimé à l'aide d'un entier. Dans notre langage, pour rester cohérent avec TSQL2, nous avons choisi d'utiliser le terme spécifique TIMEPOINT. Par exemple, TIMEPOINT'10' est un «datetime» valide. L'instant présent est représenté par le TIMEPOINT particulier NOW. Cependant, pour conserver toute la sémantique du TIMEPOINT, le NOW peut être exprimé de manière complète à l'aide du littéral TIMEPOINT.

TIMEPOINT'<int0-998>**'** | now
avec <int0-998>::= 0..998

Figure 6.2: Syntaxe d'un TIMEPOINT.

Une **période** définit une durée dans le temps. Celle-ci possède un « datetime » de départ et un « datetime » de fin [SNODGRASS, 95]. Il est possible de définir quatre types de période en fonction du choix d'inclure ou de ne pas inclure chacune des extrémités. Ces différents types de période sont appelés respectivement périodes *fermé-fermé*, *fermé-ouvert*, *ouvert-fermé* et *ouvert-ouvert*. Pour exprimer ces périodes, TSQL2 fournit quatre littéraux, le fermé à gauche « [», le fermé à droite «] », l'ouvert à gauche « (» et l'ouvert à droite «) ». Parce que nous pouvons, avec un seul type de période, exprimer toutes les autres, nous avons choisi de travailler avec le type *fermé-ouvert* (figure 6.3). Notre choix est conforme au type de période de nos bases de données temporelles (période de validité et période de transaction).

$[i_1, i_2] = [i_1, i_2+1)$
$[i_1, i_2) = [i_1, i_2)$
$(i_1, i_2] = [i_1+1, i_2+1)$
$(i_1, i_2) = [i_1+1, i_2)$
avec les instants i_1, i_2 compris entre 0..998
et avec $i_1 < i_2$

Figure 6.3: Expressions de tous les types de périodes à l'aide d'une période fermé-ouvert.

PERIOD' $[i_1, i_2)$ **'**
avec les instants i_1, i_2 compris entre 0..998 ou now
et avec $i_1 < i_2$

Figure 6.4: Syntaxe d'une PERIOD.

Lorsque nous examinons les souhaits des programmeurs, deux types de périodes spéciales manquent. En effet, les programmeurs veulent être capables et c'est l'aspect le plus important de ce langage, de comparer à une période spécifique ou à un « datetime » les différentes périodes de validité ou de transaction des événements. C'est pour cette raison que deux nouveaux littéraux, le littéral `VALID` et le littéral `TRANSACTION`, sont introduits dans notre langage. Pour pouvoir travailler avec plusieurs tables, nous devons spécifier entre crochets le nom de la table dont nous désirons la période de validité ou la période de transaction.

```
VALID(<table-name>)
TRANSACTION(<table-name>)
<table-name> est le nom de la table
```

Figure 6.5: Syntaxe d'une période de validité ou d'une période de transaction.

Si un manager veut connaître l'ensemble des personnes qui se trouvaient dans sa société durant une période déterminée, notre langage doit pouvoir satisfaire sa demande. Il est donc nécessaire de permettre aux programmeurs de réaliser de nouvelles conditions, appelées *conditions temporelles*, construites à l'aide d'**opérateurs de comparaison temporels**. Grâce à l'introduction des types `TIMEPOINT` et `PERIOD`, nous pouvons définir de manière précise ces nouveaux opérateurs de la clause `where`. Les opérateurs de période d'Allen [SNODGRASS,95] et leur représentation nous fournissent le plus riche éventail d'opérateurs temporels que nous pouvons trouver pour comparer deux périodes (figure 6.6).

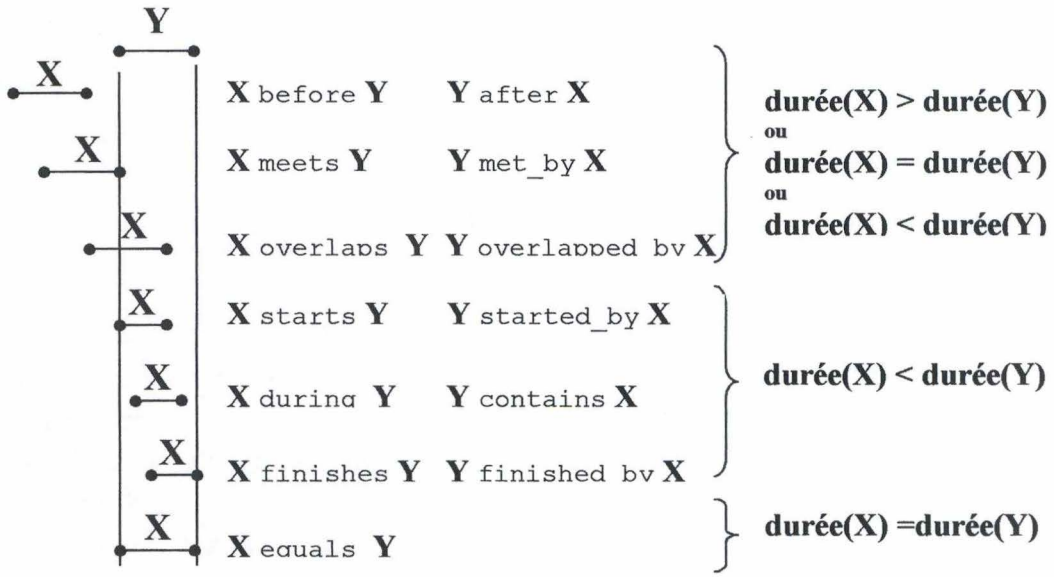


Figure 6.6: Opérateurs d'Allen.

Comme dans TSQL2, nous allons réduire de moitié l'ensemble des opérateurs d'Allen. En effet, un seul opérateur par couple d'opérateurs, excepté `equals`, est suffisant pour exprimer toutes les situations possibles entre deux périodes. Par exemple, si nous ne possédons que l'opérateur `before`, Y `after` X peut être exprimé par X `before` Y.

L'ensemble des opérateurs TSQL2 diffère des opérateurs d'Allen encore pour trois raisons.

- TSQL2 ne fournit pas les opérateurs `starts`, `started_by`, `finishes` et `finished_by`.

- Les opérateurs comme `contains` et `overlaps` ont une portée plus générale. En effet, l'opérateur `contains` peut s'effectuer entre deux périodes de même durée et l'opérateur `overlaps` de TSQL2 équivaut aux opérateurs `overlaps` et `overlapped_by` d'Allen.

- L'opérateur `before` est remplacé par l'opérateur `precedes`.

Pour notre langage, nous avons choisi d'implémenter les opérateurs de TSQL2 auxquels nous avons ajouté les opérateurs `starts` et `finishes` d'Allen (figure 6.7). Cependant, nous les utiliserons sans aucune condition sur la durée des périodes.

```
PERIOD'[i1,i2)'    PRECEDES   PERIOD'[i3,i4)'

PERIOD'[i1,i2)'    EQUALS     PERIOD'[i3,i4)'

PERIOD'[i1,i2)'    MEETS      PERIOD'[i3,i4)'

PERIOD'[i1,i2)'    CONTAINS   PERIOD'[i3,i4)'

PERIOD'[i1,i2)'    OVERLAPS   PERIOD'[i3,i4)'

PERIOD'[i1,i2)'    STARTS     PERIOD'[i3,i4)'

PERIOD'[i1,i2)'    FINISHES   PERIOD'[i3,i4)'
```

Figure 6.7: Opérateurs de comparaison temporels (period-to-period).

La clause `select` du langage mini-TSQL2

Le seul changement apporté à la clause `select` réside dans le moyen pour le programmeur de demander l'incorporation ou non dans le résultat de la période de validité ou/et de transaction. A cet effet, nous avons ajouté à notre langage un nouveau littéral. Le littéral `snapshot`, comme dans TSQL2, signifiera que nous désirons une table non temporelle comme résultat. En l'absence de ce mot, la requête fournira la table temporelle résultat adéquate. Par exemple, si `PERSONNE` est une table bitemporelle, la requête 1 de la figure 6.8 sélectionnera les colonnes `NOM`, `SALAIRE`, `V_DEBUT`, `V_FIN`, `T_DEBUT` et `T_FIN` alors que la requête 2 sélectionnera seulement les colonnes `NOM` et `SALAIRE`.

requête1

```
select NOM,SALAIRE
from   PERSONNE
where...
```

requête2

```
select snapshot NOM,SALAIRE
from   PERSONNE
where...
```

Figure 6.8: Exemple d'utilisation du littéral `snapshot`.

Le choix d'une voie explicite pour ne pas inclure dans la table résultat la période de validité ou/et de transaction nous permet d'avoir comme valeur par défaut le maintien de celles-ci. Cette syntaxe a pour défaut de ne pas permettre pour une table bitemporelle de sélectionner uniquement la période de validité ou de transaction. Néanmoins, le fait d'opérer de cette façon permet d'éviter à l'utilisateur de faire certaines requêtes dénouées de sens.

Lorsque nous introduisons le **coalescing** dans notre langage, la clause `select` se voit investie d'un nouveau rôle. Pour tout langage temporel, le coalescing est l'opération la plus délicate mais aussi la plus importante à intégrer. TSQL2 utilise la clause `from` pour donner la possibilité aux programmeurs de désigner, pour chaque table, les colonnes sur lesquelles ils souhaitent réaliser le coalescing. TSQL2 considère le coalescing comme une pré-projection spéciale de l'ensemble des colonnes de la table sur les colonnes désignées pour le coalescing. Cette pré-projection est spéciale car elle va synthétiser les éléments qui véhiculent les mêmes informations. La syntaxe de TSQL2 est simple (figure 6.10) ; les colonnes choisies sont évoquées entre deux parenthèses et séparées par une virgule. Par exemple, si le programmeur souhaite déterminer l'ensemble des noms et des salaires distincts de la table `PERSONNE` avec leur période de temps respective, nous écrirons la requête de la figure 6.9. Si aucune des colonnes n'est spécifiée, le coalescing n'est pas réalisé.

TSQL2: requête

```
select   NOM, SALAIRE
from     PERSONNE(NOM, SALAIRE)
```

Figure 6.9: Exemple de coalescing en TSQL2.

Deux cas possible pour les tables de la clause `where` en TSQL2 :

1) <table-name> (<columns-list>) [AS <alias-table>]
   ⇨ réalise le coalescing sur <column-list>

2) <table-name>
   ⇨ pas de coalescing

Figure 6.10: Syntaxe du coalescing en TSQL2.

Le coalescing en TSQL2 pose cependant un problème de compréhension majeur. Le programmeur doit en effet bien comprendre que la clause `where` effectue déjà une pré-projection et que la projection de la clause `select` ne peut se faire que sur l'ensemble des colonnes résultant de cette pré-projection. TSQL2 réalise donc une double projection. C'est pour cette raison que nous ne devons pas oublier d'ajouter, au langage TSQL2, une contrainte spéciale. Celle-ci imposerait que chaque colonne de la projection doit être incluse dans la liste des colonnes du coalescing de la table correspondante. L'exemple de la figure 6.11 illustre cette contrainte. En effet, cette requête n'est pas correcte puisque la colonne `SALAIRE` n'est pas reprise dans l'ensemble des colonnes de la table `PERSONNE` sélectionnées pour le coalescing.

TSQL2: requête incorrecte

```
select NOM,SALAIRE
from   PERSONNE(NOM,ADRESSE)
```

Figure 6.11: Exemple de requête de coalescing incorrecte en TSQL2.

L'approche du coalescing en TSQL2 permet au programmeur de faire des requêtes d'une extrême richesse. Cependant, notre langage restera plus modeste. Dans le langage mini-TSQL2, le coalescing réalisé sur une table porte obligatoirement sur l'ensemble de ses colonnes dans la projection de la clause `select`. En éliminant la double projection, nous ne laissons aucun degré de liberté aux programmeurs quant à la désignation des colonnes du coalescing. Mais puisque le coalescing sur l'ensemble des colonnes de la projection reste l'opération la plus courante dans le domaine des bases de données temporelles, nous nous limiterons à celui-ci.

Dans notre langage mini-TSQL2, un littéral spécifique, par exemple le littéral `coalescing`, aurait pu être utilisé pour désigner, non plus dans la clause `from` mais dans la clause `select`, l'envie de réaliser le coalescing sur l'ensemble des colonnes sélectionnées. Toutefois, comme les programmeurs utilisent toujours par défaut le coalescing, il serait plus judicieux de demander à ceux-ci de spécifier uniquement sa non utilisation. Ce que réalisera le littéral `every`. Ce littéral demandera l'accès à tous les enregistrements bruts de notre base de données sans avoir à réaliser le coalescing. La figure 6.12 résume la situation.

mini-TSQL2 : requête

```
select NOM,SALAIRE          ⟹  Coalescing
from   PERSONNE

select every NOM,SALAIRE    ⟹  pas de Coalescing
from   PERSONNE
```

Figure 6.12: Coalescing en mini-TSQL2.

6.3.3 Sélection dans deux tables

Le langage mini-TSQL2 se limite à l'extraction et à la sélection de données dans deux tables. Pour réaliser le produit relationnel temporel de deux tables (temporelles ou non), le langage mini-TSQL2 reste cohérent à SQL. En d'autres termes, les deux tables dont nous désirons obtenir le produit relationnel sont notées dans la clause `from` séparées par une virgule (figure 6.13).

mini-TSQL2 : requête

```
select NOM,ADRESSE,NBR_HAB,SALAIRE
from   PERSONNE,VILLE
```

Figure 6.13: Exemple de produit cartésien en mini-TSQL2.

Cependant, le produit relationnel temporel est moins utilisé que la jointure temporelle dont il en est une spécialisation. En fait, celui-ci n'est autre qu'une jointure temporelle sans condition de jointure temporelle. Comme la jointure temporelle entre deux tables est un opérateur temporel important, notre langage mini-TSQL2 doit permettre sa réalisation. Pour ce faire, le littéral « = = » a été introduit. Celui-ci permettra dans les conditions temporelles de la clause `where` de désigner la condition de jointure temporelle. La similitude entre une jointure normale SQL et une jointure temporelle mini-TSQL2 est flagrante. Toutefois, comme nous l'avons exposé au chapitre 4.5, leur signification est différente. La figure 6.14 nous donne un exemple de jointure temporelle.

mini-TSQL2 : requête

```
select NOM,ADRESSE,NBR_HAB,SALAIRE
from   PERSONNE,VILLE
where  ADRESSE==LOCALITE
```

Figure 6.14: Exemple de jointure temporelle en mini-TSQL2.

Pour éviter de confondre deux colonnes de même nom mais appartenant à deux tables différentes, notre langage mini-TSQL2 devra permettre à l'utilisateur de faire précéder à un nom de colonne le nom de la table à laquelle elle appartient.

6.4 La syntaxe complète du langage mini-TSQL2

<selection-query> ::= SELECT [EVERY] [SNAPSHOT] <column-list>
FROM <table-list>
WHERE <temporal-condition-list> , <non-temporal-conditions>

<table-list>::= <table-name>
| <table-name> , <table-name>

<column-list>::= [<table-name>.]<column-name>
| [<table-name>.] <column-name> , <column-list>

<non-temporal-conditions> : toutes les conditions normales de SQL.

<temporal-condition-list> ::= <temporal-condition>
| [( ] <temporal-condition-list> AND <temporal-condition-list> [ )]
| [( ] <temporal-condition-list> OR <temporal-condition-list> [ )]

<temporal-condition>::= <period> PRECEDES <period>
| <period> EQUALS <period>
| <period> MEETS <period>
| <period> CONTAINS <period>
| <period> OVERLAPS <period>
| <period> STARTS <period>
| <period> FINISHES <period>
| [<table-name>.]<column-name> == [<table-name>.]<column-name>

```
<period>::= VALID(<table-name>)
          | TRANSACTION(<table-name>)
          | PERIOD'[<int0-998>,<int0-998>)'
          | TIMEPOINT'<int0-998>'
          | NOW

<table-name>, <column-name> ::= [a-zA-Z] [0-9a-zA-Z]+

<int0-998>::= [0-998]
```

6.5 La sémantique du langage mini-TSQL2

6.5.1 Généralités

Afin de bien comprendre le sens donné à nos requêtes mini-TSQL2, certaines réflexions doivent être menées sur les clauses `where` et `select`. Ensuite, nous élaborerons, à partir de ces réflexions, des procédures d'évaluation de requêtes qui reflèteront la sémantique du langage mini-TSQL2.

6.5.2 La clause `where` du langage mini-TSQL2

La sémantique de nos conditions temporelles se base essentiellement sur le concept de période. En effet, toute période de validité ou de transaction est avant tout une « `PERIOD` ». C'est pour cette raison que nous pouvons facilement transformer l'une en l'autre et inversement (figure 6.15).

```
VALID(<table-name>)          ⇔ PERIOD'[<table-name>.V_DEBUT, <table-name>.V_FIN)'

TRANSACTION(<table-name>) ⇔ PERIOD'[<table-name>.T DEBUT, <table-name>.T FIN)'
```

Figure 6.15: Transformation des périodes de validité et de transaction en périodes normales (`PERIOD`).

Notre pouvons également considérer le « datetime » `TIMEPOINT` comme une période d'un seul granule car chaque événement qui a lieu durant cette période est représenté par un seul `TIMEPOINT` (figure 6.16).

```
TIMEPOINT'i' ⇒  PERIOD'[i,i+1)'
```

Figure 6.16 : Transformation d'un `TIMEPOINT` en `PERIOD`.

Puisqu'un `TIMEPOINT` est aussi une période, l'étude de la sémantique des conditions temporelles peut se limiter aux opérateurs de comparaison de périodes. De plus, puisque dans notre modèle les périodes sont des couples d'entiers et que nous connaissons l'ensemble des opérateurs

sur les entiers, nous pouvons définir de manière précise la signification des opérateurs temporels *period-to-période* (figure 6.17).

`PERIOD'[i1,i2)'` **PRECEDES** `PERIOD'[i3,i4)'` ⇒ Booléen
Le résultat est vrai si `i2<i3`

`PERIOD'[i1,i2)'` **EQUALS** `PERIOD'[i3,i4)'` ⇒ Booléen
Le résultat est vrai si `(i1=i3)` and `(i2=i4)`

`PERIOD'[i1,i2)'` **MEETS** `PERIOD'[i3,i4)'` ⇒ Booléen
Le résultat est vrai si `i2=i3`

`PERIOD'[i1,i2)'` **CONTAINS** `PERIOD'[i3,i4)'` ⇒ Booléen
Le résultat est vrai si `(i1<=i3)` and `(i2>=i4)`

`PERIOD'[i1,i2)'` **OVERLAPS** `PERIOD'[i3,i4)'` ⇒ Booléen
Le résultat est vrai si `(i2>i3)` and `(i1<i4)`

`PERIOD'[i1,i2)'` **STARTS** `PERIOD'[i3,i4)'` ⇒ Booléen
Le résultat est vrai si `i1=i3`

`PERIOD'[i1,i2)'` **FINISHES** `PERIOD'[i3,i4)'` ⇒ Booléen
Le résultat est vrai si `i2=i4`

Figure 6.17: Opérateurs de comparaison temporels (period-to-period).

Pour rester cohérent avec SQL, une requête temporelle ne possédant aucune condition temporelle accèdera uniquement aux états courants.

6.5.3 La clause `select` du langage mini-TSQL2

La clause `select` permet à l'utilisateur d'exprimer son désir ou son refus d'opérer le coalescing sur la ou les tables de la clause `from`. La signification d'une requête mini-TSQL2 passe par une bonne compréhension de l'opérateur de coalescing. De fait, il existe deux manières de percevoir le coalescing quand celui-ci s'effectue sur une table temporelle dont les colonnes relatives à l'identifiant d'entité ne sont pas sélectionnées. Considérons la table mono-temporelle avec « valid time » `C_H_PERSONNE` de la figure 6.18.

C_H_PERSONNE				
NOM	**ADRESSE***	**SALAIRE***	**V_DEBUT**	**V_FIN**
Léo	Liège	50000	10	35
Léo	Bruxelles	100000	35	999
René	Namur	20000	15	25
René	Namur	30000	25	50
René	Namur	50000	50	75
René	Bruxelles	50000	75	90
Jean	Liége	25000	20	25
Jean	Namur	25000	25	40
Jean	Namur	60000	40	999

Figure 6.18: Table mono-temporelle avec « valid time » C_H_PERSONNE.

Si nous procédons à l'exécution d'une requête qui sélectionne l'identifiant d'entité, le coalescing ne pose aucun problème. Par exemple, nous pouvons réaliser la requête de la figure 6.19. Le résultat nous donnera pour chaque personne (entité), l'historique de son adresse. La table résultat conserve donc toutes les caractéristiques d'une table temporelle normalisée. En effet, l'historique de l'adresse de chaque entité ne possède pas de trou ni de recouvrement d'information.

Requête :

```
select NOM ADRESSE
from   C_H_PERSONNE
```

Résultat_PERSONNE			
NOM	**ADRESSE***	**V_DEBUT**	**V_FIN**
Léo	Liège	10	35
Léo	Bruxelles	35	999
René	Namur	15	75
René	Bruxelles	75	90
Jean	Liége	20	25
Jean	Namur	25	999

Figure 6.19: Exemple de requête de coalescing avec sélection de l'identifiant de l'entité et résultat de la requête.

Le problème du coalescing se pose lorsque nous effectuons une requête qui ne sélectionne pas l'identifiant d'entité. En effet, dans ce cas nous pouvons soit considérer le coalescing comme une opération qui ne peut se réaliser qu'avec l'identifiant d'entité, soit comme une opération à part qui ne se soucie pas de l'existence de cet identifiant dans la requête de sélection.

Dans le premier cas, l'opération de coalescing est réalisée comme si l'identifiant de l'entité était présent dans la requête de sélection. Toutefois, celui-ci ne figure pas dans le résultat. La figure 6.20 illustre cette démarche.

Requête :

```
select ADRESSE
from   C_H_PERSONNE
```

Résultat_PERSONNE		
ADRESSE	**V_DEBUT**	**V_FIN**
Liège	10	35
Bruxelles	35	999
Namur	15	75
Bruxelles	75	90
Liége	20	25
Namur	25	999

Figure 6.20: Exemple de requête de coalescing sans sélection de l'identifiant d'entité mais en considérant son existence.

La signification de la table `Résultat_PERSONNE` est non triviale. Elle enregistre chaque adresse ainsi que la période durant laquelle **une** personne (entité) particulière y a habité. Si deux individus habitait à la même adresse durant la même période, deux enregistrements se retrouveraient dans la table `Résultat_PERSONNE`. Le résultat est donc non normalisé car il peut y avoir des trous ainsi que des redondances d'information.

La seconde solution consiste à considérer le coalescing, quelle que soit l'entité d'où provient l'information, comme un simple opérateur de synthèse d'enregistrements, la figure 6.21 illustre cette solution.

Requête :

```
select ADRESSE
from   C_H_PERSONNE
```

Résultat_PERSONNE		
ADRESSE	**V_DEBUT**	**V_FIN**
Bruxelles	35	999
Liège	10	35
Namur	15	999

Figure 6.21: Exemple de requête de coalescing sans sélection de l'identifiant d'entité et sans considérer son existence.

La table résultat de la figure 6.21 nous renseigne sur les périodes durant lesquelles il y a eu **au moins une** personne à une adresse déterminée. Cependant, le résultat reste non normalisé car il peut y avoir des trous d'information entre les périodes d'une même adresse.

Le choix entre ces deux perceptions du coalescing doit être établi clairement avant toute implémentation. En ce qui nous concerne, nous opterons pour la première solution.

6.5.4 Interprétation d'une requête mini-TSQL2

Les requêtes sur une table

Les requêtes mini-TSQL2 étant relativement complexes, nous allons proposer une procédure d'évaluation qui permettra d'interpréter leur signification.

Procédure d'évaluation :

- on considère la table spécifiée dans la clause `from` ;
- on sélectionne les lignes respectant les conditions non temporelles de la clause `where` ;
- on extrait les valeurs demandées par la clause `select` en considérant les différentes périodes;
- si le littéral `every` n'apparaît pas dans la requête, on réalise le coalescing ;
- on sélectionne les lignes respectant les conditions temporelles de la clause `where`.
- on considère le littéral `snapshot`.

Pour être complet dans notre exposé, nous devons justifier la place des conditions non temporelles et temporelles dans notre procédure d'évaluation.

Comme la table résultant du coalescing n'est composée que des colonnes sélectionnées par la requête, les conditions non temporelles doivent être considérées avant l'étape du coalescing. En effet, toutes les conditions non temporelles portant sur d'autres colonnes n'ont plus aucune signification lorsqu'on les considère après le coalescing.

Les conditions temporelles interviennent après l'étape du coalescing et agissent sur la table qui en résulte. Cette position dans la procédure d'évaluation résulte de l'interprétation que nous portons au coalescing et aux concepts de périodes de validité et de transaction. Reprenons la table de la figure 6.18 et considérons la requête de la figure 6.22.

```
select NOM,ADRESSE
from   C_H_PERSONNE
where  valid(C_H_PERSONNE) contains period'[35,60)
```

Figure 6.22: Requête temporelle.

Le programmeur peut comprendre de deux façons une telle requête. En effet, la condition temporelle peut être perçue comme une condition sur les enregistrements non synthétisés ou sur les enregistrements préalablement synthétisés. Cela correspond à la prise en compte des conditions temporelles respectivement avant le coalescing et après celui-ci. Comme le montrent les figures 6.23 et 6.24, les résultats de la requête obtenus selon ces deux interprétations sont totalement différents.

Résultat_PERSONNE			
NOM	**ADRESSE***	**V_DEBUT**	**V_FIN**
Léo	Bruxelles	35	999

Figure 6.23: Résultat de la requête de la figure 6.22 lorsque les conditions temporelles agissent sur des enregistrements non synthétisés.

Résultat_PERSONNE			
NOM	**ADRESSE***	**V_DEBUT**	**V_FIN**
Léo	Bruxelles	35	999
René	Namur	15	75
Jean	Namur	25	999

Figure 6.24: Résultat de la requête de la figure 6.22 lorsque les conditions temporelles agissent sur des enregistrements synthétisés.

Dans notre implémentation, nous avons opté pour la prise en compte des conditions temporelles après le coalescing. En effet, la philosophie de notre implémentation est simple. Nous voulons fournir à l'utilisateur avant toute condition temporelle, la table qu'il aurait obtenue si celle-ci ne possédait que les informations des colonnes désirées dans la requête. Dans notre exemple, si notre table n'avait pas enregistré la variation des salaires, les enregistrements de celle-ci aurait eu pour périodes de validité celles déterminées par le coalescing. C'est donc pour cette raison que nous avons opté pour une prise en compte des conditions temporelles après le coalescing.

Cependant, pour des raisons de puissance de notre langage, nous devons nous assurer qu'il sera toujours possible d'obtenir, à l'aide de requêtes effectuant les conditions temporelles après le coalescing, un résultat équivalent à la réalisation des conditions temporelles avant le coalescing. Ce résultat est possible à l'aide de deux requêtes. La première réalisera une requête temporelle sans coalescing mais qui tiendra compte des conditions temporelles. La seconde accomplira le coalescing sur le résultat de la première requête. La figure 6.25 illustre cette méthode pour la requête de la figure 6.22.

1° Requête

```
insert into TABLE_TEMP
select every NOM,ADRESSE,SALAIRE
from   C_H_PERSONNE
where  valid(C_H_PERSONNE) contains period'[35,60)
```

2° Requête

```
select NOM,ADRESSE
from   TABLE_TEMP
```

Figure 6.25: Exemple de réalisation d'une requête où les conditions temporelles sont prises en compte avant le coalescing alors que l'implémentation prend en compte les conditions temporelles après le coalescing.

Les requêtes sur deux tables

L'interprétation des requêtes mini-TSQL2 sur deux tables est très compliquée. Celle-ci peut néanmoins être exprimée à l'aide d'une procédure d'évaluation.

Procédure d'évaluation :

- on considère les tables spécifiées dans la clause `from` ;
- pour chacune des tables :
  - on extrait les valeurs demandées par la clause `select` en considérant les différentes périodes ;
  - si le littéral `every` n'apparaît pas dans la requête, on réalise le coalescing ;
- on effectue la jointure temporelle en tenant compte des conditions non temporelles et temporelles de la clause `where` ;
- on considère le littéral `snapshot`.

Ce type de procédure d'évaluation soulève quelques questions. En effet, avant de faire la jointure temporelle, le coalescing est réalisé sur chacune des deux tables. Est-ce que la jointure de deux tables « coalescées » équivaut au coalescing de la jointure de deux tables ? Considérons deux tables mono-temporelles avec « valid time »: `TAB1` et `TAB2`. Supposons que `TAB1` possède une clé étrangère temporelle vers `TAB2`. Si, à l'instar de notre traducteur, nous opérons le coalescing sur chacune des deux tables, nous pouvons certifier que chaque entité de celles-ci ne possède pas deux enregistrements consécutifs dans le temps véhiculant la même information. La figure 6.26 illustre cette constatation sur deux entités particulières, une de chaque table. Ax représente l'information

véhiculée par un enregistrement de l'entité A entre les moments t_x et t_{x+1}. De même, By représente l'information véhiculée par un enregistrement de l'entité B entre le moment $t_{y'}$ et $t_{y'+1}$.

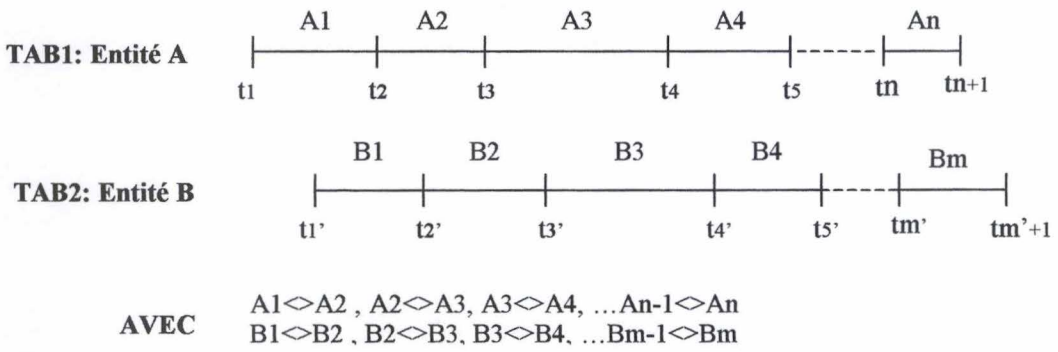


Figure 6.26: Exemple de situation de deux tables après que chacune ait subi le coalescing.

Supposons maintenant que l'enregistrement A3 de `TAB1` référence l'entité B de `TAB2`. Dans ce cas, si nous appliquons les règles de la jointure temporelle de deux tables mono-temporelles avec valid time, nous obtenons un ensemble d'enregistrements consécutifs qui ne véhiculent pas la même information (figure 6.27). Cet exemple est généralisable à tous les enregistrements de l'entité ainsi qu'à toutes les entités. Nous pouvons donc conclure que le résultat obtenu par cette jointure temporelle est déjà synthétisé.

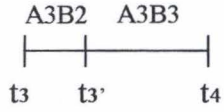


Figure 6.27: Résultat de la jointure temporelle entre l'enregistrement A3 et les enregistrements de l'entité B.

Supposons maintenant que les tables `TAB1` et `TAB2` soient jointes avant d'être synthétisées. A3 aurait pu, par exemple, avant la jointure être composé de deux enregistrements A31 et A32 consécutifs et véhiculant la même information ainsi que B2 avec B21 et B22 et que B3 avec B31 et B32 (figure 6.28).

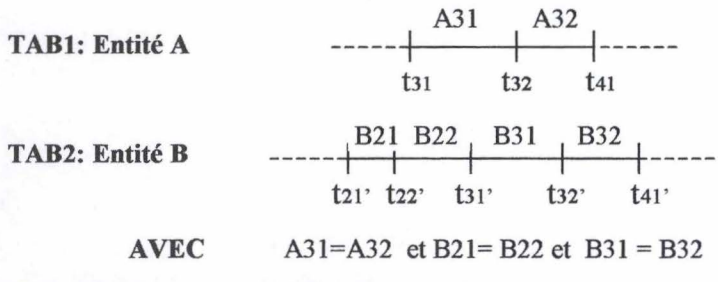
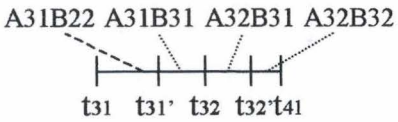


Figure 6.28: Exemple de situation de deux tables avant le coalescing.

La figure 6.29 illustre la jointure temporelle effectuée entre les enregistrements A31 et A32 et les enregistrements de l'entité B.



AVEC A31B31=A32B31=A32B32

Figure 6.29: Jointure temporelle des enregistrements A31 et A32 avec les enregistrements de B.

A présent, si nous opérons le coalescing sur le résultat de cette jointure temporelle (figure 6.30), nous obtenons exactement le même résultat que la figure 6.27.

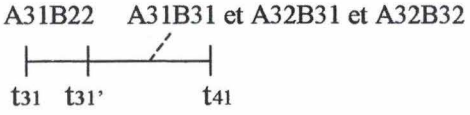


Figure 6.30: Coalescing sur la jointure temporelle des enregistrements A31 et A32 avec les enregistrements de B.

De la même manière, nous pouvons démontrer théoriquement, pour tous les cas de figure, que le coalescing avant ou après la jointure temporelle donne un résultat équivalent (figure 6.31).

$$C(TAB1) \times C(TAB2) \Longleftrightarrow C(TAB1 \times TAB2)$$

C( ) : opérateur de coalescing $\times$: jointure temporelle

Figure 6.31: Equivalence du coalescing avant et après la jointure temporelle.

La procédure d'évaluation que nous sommes en train de construire pose certains problèmes majeurs. Contrairement aux requêtes appliquées sur une seule table, les conditions non temporelles interviennent lors de la requête de jointure temporelle finale et non avant le coalescing. En fait, comme certaines conditions non temporelles portent sur les deux tables, il est impossible de les faire intervenir, lors du coalescing, sur chacune de celles-ci. Cependant, un problème existe dans cette interprétation. Les requêtes possédant certaines conditions non temporelles ne sont pas réalisables. En effet, lorsque celles-ci portent sur une ou plusieurs colonnes ne figurant pas dans l'ensemble des colonnes sélectionnées par la requête, notre traducteur est incapable de les réaliser. Ceci est la conséquence de l'exécution du coalescing avant la prise en compte des conditions non temporelles. Le coalescing génère deux nouvelles tables qui sont composées uniquement des colonnes de la sélection. Cela explique l'incapacité d'effectuer une requête possédant une condition non temporelle sur des colonnes non sélectionnées. Ce même problème peut être observé pour la condition de jointure temporelle. En effet, ce problème apparaît lorsque la requête ne sélectionne pas les colonnes relatives à la condition de jointure temporelle.

Ce problème peut être résolu en réalisant le coalescing après la jointure temporelle et en tenant compte, lors de celle-ci, des conditions temporelles et non temporelles. Ceci nous est permis grâce à l'équivalence des résultats entre le coalescing réalisé avant ou après la jointure temporelle. La procédure d'évaluation suivante illustre ce type d'interprétation. Dans celui-ci, plus aucune restriction n'existe sur les conditions non temporelles car celles-ci agissent sur la jointure des tables possédant l'ensemble des colonnes.

Procédure d'évaluation :

- on considère les tables spécifiées dans la clause `from` ;
- on effectue la jointure temporelle en tenant compte des conditions non temporelles et temporelles de la clause `where`.
- on extrait les valeurs demandées par la clause `select` en considérant les différentes périodes
- si le littéral `every` n'apparaît pas dans la requête, on réalise le coalescing ;
- on considère le littéral `snapshot`

Toutefois, lorsque le coalescing est effectué après la jointure temporelle, un autre problème se présente. Celui-ci concerne les conditions temporelles. En effet, celles-ci agissent maintenant sur les tables qui ne sont pas encore synthétisées ou « coalescées ». Comme nous l'avons déjà souligné pour les requêtes appliquées sur une table, la sémantique de nos conditions temporelles nous impose de réaliser celles-ci après la phase de coalescing, ce qui n'est pas le cas dans notre type d'interprétation. Pour pallier cette erreur, nous pourrions penser qu'il suffit d'effectuer les conditions temporelles après le coalescing. Cependant, nos conditions temporelles agissent sur les différentes périodes des tables de la requête et non pas sur la période résultant de la jointure temporelle des deux tables. Par exemple, considérons deux tables `TAB1` et `TAB2` et `la` condition temporelle `VALID(TAB1) OVERLAPS PERIOD'[x,y)'` qui sélectionne les enregistrements de la table `TAB1` compris entre deux «datetimes» particuliers. Cette condition temporelle ne peut pas être appliquée après le coalescing sur le résultat de la jointure temporelle car celui-ci n'enregistre plus la période des enregistrements de `TAB1` mais la période résultant de la jointure des tables `TAB1` et `TAB2`. En résumé, si le coalescing est réalisé après la jointure temporelle, on est obligé de tenir compte des conditions temporelles lors de la jointure. Or, comme nous l'avons déjà souligné, les conditions temporelles doivent intervenir après le coalescing pour préserver la sémantique de notre langage mini-TSQL2 ; ce qui est impossible.

En conclusion, nous allons opter pour la procédure d'évaluation effectuant le coalescing avant la jointure temporelle. Dans ce cas, toutes les conditions temporelles sont réalisables. Il existe, néanmoins, des restrictions portant sur les conditions non temporelles. Celles-ci varient selon la position occupée par les conditions dans le processus d'évaluation. Si les conditions non temporelles sont considérées avant le coalescing, chacune de celles-ci doit porter uniquement sur les colonnes d'une des deux tables. Par contre, si les conditions non temporelles sont considérées lors de la jointure temporelle, seules les conditions portant sur les colonnes sélectionnées par la requête sont réalisables.

Comme ces différentes restrictions ne sont pas implicites, nous pourrions restreindre notre langage mini-TSQL2 à la réalisation d'une simple jointure temporelle. Celle-ci ne possèderait ni condition non temporelle ni condition temporelle. Seule la condition de jointure serait présente. Nous laisserions donc le soin à l'utilisateur d'effectuer les conditions voulues avant ou après la requête de jointure mini-TSQL2.

Pour terminer, une remarque importante reste à formuler. L'implémentation du colaescing après la jointure temporelle aurait pu être envisagée si nous nous étions limité à des conditions temporelles plus simples. En effet, au lieu de permettre comme dans TSQL2 la désignation du « valid time » et du « transaction time » sur chacune des deux tables, nous aurions pu considérer ses périodes comme celles résultant de la jointure. Au niveau syntaxique, les littéraux `valid` et `transaction` sans aucun nom de table auraient suffi pour désigner respectivement le « valid time » et le « transaction time » du résultat de la jointure de deux tables. Cependant, dans ce cas, la sémantique de notre langage en aurait été modifiée.

6.6 Respect des desiderata des utilisateurs

Afin de vérifier le respect des desiderata des utilisateurs (paragraphe 6.3.1), nous allons passer en revue chacun de ceux-ci et montrer qu'il est possible de les réaliser à l'aide du langage mini-TSQL2. Tous les exemples de ce chapitre utiliseront une table (TABLEX) mono-temporelle avec « valid time » à n colonnes dont les m premières seront non temporelles tandis que les n-m dernières seront temporelles.

- **Extraction des états courants de toutes les entités.**

Ce genre de requête est très simple à réaliser avec le langage mini-TSQL2 car sans condition temporelle, le langage accède automatiquement aux états courants.

Exemple:

```
select snapshot COL1,COL2...COLn
from   TABLEX
```

- **Extraction des états de toutes les entités à un instant T donné (dans le passé ou dans le futur).**

Pour réaliser cette requête, l'opérateur `contains` sera utilisé.

Exemple:

```
select snapshot COL1,COL2...COLn
from   TABLEX
where  valid(TABLEX) contains timepoint'T'
```

- **Extraction de l'historique de toutes les entités durant une période déterminée [T1,T2).**

Pour réaliser cette requête, l'opérateur `overlaps` sera utilisé.

Exemple:

```
select COL1,COL2...COLn
from   TABLEX
where  valid(TABLEX) overlaps period'[T1,T2)'
```

- **Historique partiel des entités.**

L'historique est partiel lorsque nous ne sélectionnons que certaines colonnes temporelles. Supposons que nous sélectionnons les j premières colonnes, avec m<j<n.

Exemple:

```
select COL1,COL2...COLj
from   TABLEX
```

6.7 Exemples de requêtes

Les exemples de requêtes ont été élaborés à partir des tables `PERSONNE_1`, `PERSONNE_4`, `PERSONNE` et `VILLE` des figures 4.1, 4.3 et 4.17.

Sur la table `PERSONNE_1` de la figure 4.1

Requête textuelle: Quelles sont les personnes qui ont habité à Namur entre le 20 et le 30?
Requête mini-TSQL2:

```
select snapshot NOM
from   PERSONNE_1
where  valid(PERSONNE_1) overlaps period'[20,30)',
       Adresse = 'Namur'
```

Réponse: {René, Jean}

Requête textuelle: Quelles sont les personnes qui ont habité à Namur durant la période du 20 au 30?
Requête mini-TSQL2:

```
select snapshot NOM
from   PERSONNE_1
where  valid(PERSONNE_1) contains period'[20,30)',
       Adresse = 'Namur'
```

Réponse: {René}

Requête textuelle : Quelles étaient, le 18, les adresses et les noms des personnes dont le salaire était supérieur à 25.000 francs?

Requête mini-TSQL2 :

```
select snapshot NOM,ADRESSE
from   PERSONNE_1
where  valid(PERSONNE_1) contains timepoint'18',
       SALAIRE > 25000
```

Réponse : {(Léo, Liège)}

Requête textuelle : Quel est l'historique des salaires de René après le 20?

Requête mini-TSQL2:

```
select SALAIRE
from   PERSONNE_1
where  timepoint'20' precedes valid(PERSONNE_1),
       NOM='René'
```

Réponse : {(30000, 25, 50) ; (50000, 50, 90)}

Requête textuelle : Quel est l'historique des salaires de René après le 20 (sans coalescing)?

Requête mini-TSQL2:

```
select every SALAIRE
from   PERSONNE_1
where  timepoint'20' precedes valid(PERSONNE_1),
       NOM='René'
```

Réponse : {(30000, 25, 50) ; (50000, 50, 75) ; (50000, 75, 90)}

Sur la table PERSONNE_4 de la figure 4.3.

Requête textuelle : Quelles étaient nos croyances, le 35, à propos de l'adresse et du salaire de Léo entre le 15 et le 22 ?

Requête mini-TSQL2:

```
select snapshot ADRESSE,SALAIRE
from   PERSONNE_4
where  transaction(PERSONNE_4) contains timepoint'35'
and    valid(PERSONNE_4) overlaps period'[15,22)',
       NOM='Léo'
```

Réponse : {(Namur, 60000); (Liège,60000)}

Sur les tables PERSONNE et VILLE de la figure 4.17.

La table personne sera considérée comme une table mono-temporelle avec « valid time ».

Requête textuelle : Quel était l'historique des adresses et du nombre d'habitants de la ville de Léo entre le 20 et le 25?

Requête mini-TSQL2:

```
select NOM,ADRESSE,NBR_HAB
from   PERSONNE,VILLE
where  valid(PERSONNE) overlaps period'[20,25)'
and    ADRESSE==LOCALITE ,
       NOM='Léo'
```

Réponse : {(Léo, Liège, 450,10,30); (Léo, Liège, 500,30,35)}

6.8 Puissance du langage mini-TSQL2

L'algèbre de Codd pour les bases de données relationnelles fût la première algèbre créée pour un modèle de bases de données. Cette algèbre est basée sur le concept de relations et possède un ensemble d'opérateurs relationnels. Les opérateurs fondamentaux sont: l'*union*, la *différence*, le *produit cartésien*, la *projection* et la *sélection*. Par la suite, d'autres opérateurs sont venus s'ajouter comme l'*intersection*, la *division* et la *jointure*. Cependant, aucun de ces nouveaux opérateurs n'a ajouté de la puissance à l'algèbre de Codd car ceux-ci peuvent tous être exprimés à l'aide des cinq opérateurs de base. Par exemple, l'intersection n'est qu'un raccourci pour deux différences particulières : $R \cap S = R-(R-S)$. Grâce à cette algèbre, on pourrait spécifier les données que l'on veut extraire de la base de données. Toutefois, l'utilisation de cette algèbre comme un langage de requêtes est plutôt difficile car il faut que l'utilisateur connaisse parfaitement toutes les relations de la base de données pour pouvoir naviguer entre celles-ci. C'est en résolvant ce problème et en permettant d'exprimer les requêtes sous une forme déclarative que le calcul relationnel a été introduit. Le calcul relationnel est un langage logique à partir duquel la majorité des langages relationnels sont construits.

Une des caractéristiques fondamentales de la théorie relationnelle réside dans le fait que l'algèbre relationnelle et le calcul relationnel ont la même puissance. Cela signifie que chaque résultat d'un calcul relationnel peut être obtenu par une expression algébrique et inversement. Maintenant, l'algèbre relationnelle et le calcul relationnel sont considérés comme des références dans le monde des modèles de bases de données. C'est pourquoi, lorsqu'un nouveau langage relationnel est créé, l'auteur de celui-ci s'empresse de démontrer que ce langage possède la même puissance que l'algèbre relationnelle ou que le calcul relationnel. Par exemple, SQL possède, pour l'essentiel, la même puissance que l'algèbre relationnelle.

Le problème avec le modèle relationnel traditionnel est qu'il ne supporte pas les aspects temporels du monde réel. L'algèbre relationnelle de Codd a donc besoin d'être étendue pour traiter des aspects temporels. C'est ce que fit, par exemple, Snodgrass avec une algèbre dédiée à TSQL2 [SNODGRASS, 95]. La caractéristique principale de cette algèbre est que la plupart de ses opérateurs sont de simples extensions d'opérateurs non temporels existants. Quant à la puissance du langage TSQL2, elle est presque équivalente à son algèbre. En effet, Snodgrass a défini TSQL2 avant d'étendre l'algèbre relationnelle. C'est pourquoi quelques caractéristiques du langage ne sont pas encore supportées par l'algèbre dédiée à TSQL2.

Pour déterminer la puissance de notre langage mini-TSQL2, il faudrait pouvoir comparer celui-ci au langage TSQL2. Toutefois, ces deux langages ne sont comparables qu'en prenant quelques précautions. En effet, notre langage mini-TSQL2 fonctionne sur un modèle de données particulier (chapitre 3) plus restrictif que le modèle de données de TSQL2. Les principales différences entre ces deux modèles sont exposés à la figure 6.32. Le modèle de données de TSQL2 doit donc être réduit pour permettre une comparaison des deux langages. Cette restriction sera dénommée TSQL2bis. Il nous appartient donc maintenant de déterminer si notre langage possède la

même puissance que TSQL2bis. En d'autres termes, est-il possible de réaliser avec notre langage tout ce que TSQL2bis permet.

Modèle de données du langage mini-TSQL2	Modèle de données du langage TSQL2
- Utilisation d'une seule granularité - Continuité des périodes d'une entité - Pas d'indétermination temporelle	- Utilisation de plusieurs granularités - Possibilité de trous dans les périodes d'une entité - Supporte l'indétermination temporelle

Figure 6.32: Principales différences entre le modèle de données du langage mini-TSQL2 et celui de TSQL2.

Une démonstration théorique serait trop difficile à réaliser. D'ailleurs, [SNODGRASS, 95] affirme qu'aucune étude théorique rigoureuse n'a été effectuée jusqu'à présent pour évaluer la puissance d'expression du langage TSQL2. Nous devons donc rester pragmatique dans notre approche. Pour ce faire, nous allons parcourir l'ensemble des clauses des requêtes TSQL2 et mini-TSQL2 en vue de souligner leurs différences. Chacune de celles-ci sera examinée afin de déterminer une différence de puissance quelconque entre les deux langages.

La clause `from`

Différence 1 :

- Le langage TSQL2 permet l'extraction et la sélection de données dans plusieurs tables.
- Le langage mini-TSQL2 est limité à l'extraction et à la sélection de données dans deux tables.

Comme pour SQL, cette limitation ne diminue en rien la puissance de notre langage puisque les requêtes portant sur plusieurs tables peuvent être réalisées en plusieurs étapes à l'aide de requêtes sur deux tables. La figure 6.33 illustre cette affirmation en SQL. Dans cette exemple, le numéro des colonnes correspond au numéro de la table à laquelle elles appartiennent.

```
select COL1,COL2,COL3...
from   TABLE1,TABLE2,TABLE3
```

⟺

```
insert into TABLE_TEMP
select COL1,COL2
from   TABLE1,TABLE2

select COL1,COL2,COL3
from   TABLE_TEMP,TABLE3
```

Figure 6.33: Equivalence, en SQL, entre le produit cartésien de trois tables et deux produits cartésiens de deux tables.

Différence 2 :

- En TSQL2, le coalescing d'une table est réalisé sur l'ensemble des colonnes désignées à cet effet dans la clause `from`. Grâce à cette syntaxe, l'utilisateur peut effectuer une double projection.
- En mini-TSQL2, le coalescing d'une table est réalisé sur l'ensemble de ses colonnes dans la projection de la clause `select`.

Deux requêtes mini-TSQL2 suffisent pour obtenir la double projection permise par la syntaxe du coalescing en TSQL2. La première effectue le coalescing tandis que la seconde réalise la sélection désirée. La figure 6.34 illustre cette possibilité sur une requête générique.

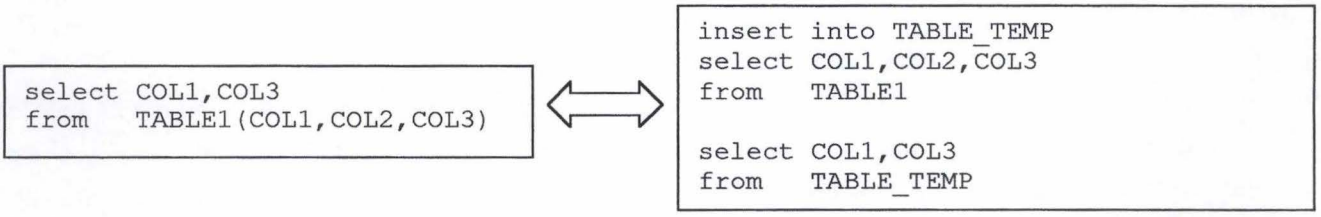


Figure 6.34: Equivalence entre la double projection permise par la syntaxe du coalescing en TSQL2 et deux requêtes mini-TSQL2.

Différence 3 :

- En TSQL2, le coalescing n'est pas effectué sur une table si aucune de ses colonnes ne sont mentionnées dans la clause `from` de la requête.
- En mini-TSQL2, le coalescing n'est pas effectué sur une table si le littéral `every` apparaît dans la clause `select`.

La puissance de notre langage est cependant altérée par ce changement de syntaxe. En effet, le littéral `every` porte sur l'ensemble des tables de la clause `from`, ce qui limite notre langage. Il nous est donc impossible de demander la réalisation du coalescing sur l'une des deux tables composant la clause `from` d'une requête mini-TSQL2 complexe.

La clause `where`

Différence 1 :

Les différences majeures des clauses `where` TSQL2 et mini-TSQL2 sont essentiellement de type syntaxique. La figure 6.35, où x et y représentent deux périodes, donne pour chaque opérateur *period-to-period* TSQL2 son équivalent mini-TSQL2.

TSQL2	mini-TSQL2
x = y	x EQUALS y
x PRECEDES y	x PRECEDES y
x OVERLAPS y	x OVERLAPS y
x CONTAINS y	x CONTAINS y
x MEETS y	x MEETS y
BEGIN(x) = BEGIN(y)	x STARTS y
END(x) = END(y)	x FINISHES y

Figure 6.35: Equivalence des opérateurs *period-to-period* de TSQL2 et mini-TSQL2.

Cependant, le langage mini-TSQL2 est moins puissant que TSQL2 car il ne possède pas d'opérateur de construction et de destruction de périodes. En TSQL2, ces opérateurs sont `PERIOD`, `INTERSECT`, `BEGIN` et `END`. La définition de ces différents opérateurs se trouve dans [SNODGRASS, 95] (figure 6.36).

Opérateurs	Explications
`period ← PERIOD(datetime1,datetime2)`	Renvoit la période [datetime1, datetime2) si celle-ci est valide, NULL sinon.
`period ← INTERSECT (period1,period2)`	Renvoit l'intersection entre période1 et période2 si elle existe, NULL sinon.
`datetime ← BEGIN (period)`	Renvoit le datetime délimitant le début de la période.
`datetime ← END (period)`	Renvoit le datetime délimitant la fin de la période.

Figure 6.36: Opérateurs TSQL2 de construction et de destruction de périodes.

Différence 2 :

- Le langage TSQL2 accepte les sous-requêtes temporelles.
- Le langage mini-TSQL2 n'accepte pas les sous-requêtes temporelles.

Comme pour SQL, les sous-requêtes permettent une simplification du langage mais n'améliorent pas la puissance de celui-ci. En effet, pour réaliser, en mini-TSQL2, une structure de requêtes emboîtées, il suffit d'extraire les résultats des sous-requêtes avant de les utiliser dans la construction de la requête principale.

La clause `select`

Différence 1 :

- Le langage TSQL2 possède des fonctions d'agrégation.
- Le langage mini-TSQL2 ne possède aucune fonction d'agrégation.

Cette différence illustre la pauvreté de la clause `select` du langage mini-TSQL2. Ce manque d'expressivité porte donc préjudice à la puissance de notre langage.

Différence 2 :

- Le langage TSQL2 permet la projection du « valid time » et du « transaction time ».
- Le langage mini-TSQL2 ne permet pas la projection du « valid time » et du « transaction time ».

Le langage mini-TSQL2 permet uniquement à l'utilisateur l'incorporation du « valid time » ou du « transaction time » dans le résultat d'une requête temporelle. Toutefois, TSQL2 va plus loin en fournissant à l'utilisateur deux nouvelles clauses : la clause valid et la clause transaction. Celles-ci permettent respectivement la projection des périodes du « valid time » et du « transaction time ». Cela signifie que les périodes issues d'une base de données peuvent être retravaillées à l'aide d'expressions temporelles. Ces deux clauses sont très utiles lorsque de nouvelles informations qui doivent être introduites dans une base de données sont calculées à partir d'informations existantes. Notre langage mini-TSQL2 est donc moins puissant que TSQL2.

Les clauses `group by`, `having` et `order by`

Différence 1 :

- Le langage TSQL2 permet les clauses `group by`, `having` et `order by`.
- Le langage mini-TSQL2 ne permet pas les clauses `group by`, `having` et `order by`.

Conclusions

En conclusion, le langage mini-TSQL2 est moins puissant que le langage TSQL2 essentiellement à cause de cinq lacunes :

- l'impossibilité de demander la réalisation du coalescing sur l'une des deux tables composant la clause `from` d'une requête mini-TSQL2 complexe,
- l'absence d'opérateur de construction et de destruction de périodes,
- l'absence de fonctions agrégatives,
- l'impossibilité de réaliser une projection du « valid time » et du « transaction time »,
- l'absence de clauses `group by`, `having` et `order by`.

Cependant, malgré sa simplicité, le langage mini-TSQL2 couvre largement les besoins de l'utilisateur.

7
Implémentation du langage temporel mini-TSQL2

7.1 Introduction

La recherche que nous avons menée a pour objectif de déboucher sur un prototype d'implémentation du langage temporel mini-TSQL2. Lorsqu'on désire implémenter un nouveau langage, on est confronté au choix de créer un nouveau SGBD qui supporte ce langage ou de créer un traducteur. Ce dernier traduit les requêtes du nouveau langage vers un langage existant propre à un SGDB. Ce type de démarche est souvent utilisé en informatique et porte le nom de « pont » (bridge, en anglais). Pour notre implémentation, nous avons opté pour un « pont » entre le langage mini-TSQL2 et le langage SQL qui est universellement connu. La figure 7.1 illustre notre démarche.

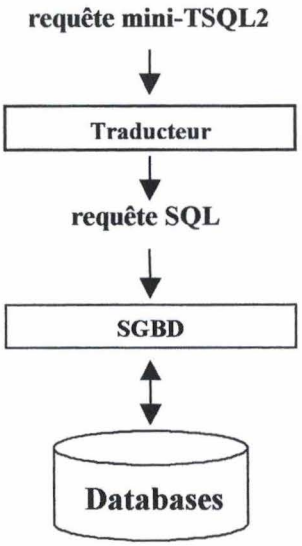


Figure 7.1: Schéma de l'exécution d'une requête mini-TSQL2 traduite en requête SQL.

La première partie de l'implémentation de notre langage mini-TSQL2 consiste à définir et à implémenter un module de traduction d'une requête mini-TSQL2 en une requête SQL. Le paragraphe 7.2 détaille les différentes étapes de l'implémentation de ce module. Afin d'être clairs, comme pour la présentation de notre langage au chapitre précédent, nous présenterons tout d'abord un traducteur pour les requêtes d'accès à une seule table (paragraphe 7.2.1) pour ensuite évoquer le problème des requêtes à deux tables (paragraphe 7.2.2). Ensuite, le module de traduction sera utilisé pour implémenter une interface de programmation qui permettra à tout programmeur d'émettre une requête mini-TSQL2 à partir d'un programme (paragraphe 7.3). Le paragraphe 7.3.3 définit et implémente cette interface de programmation. Pour réaliser celle-ci, nous utiliserons ODBC. Le paragraphe 7.3.1 en expliquera la raison tandis que le paragraphe 7.3.2 rappellera sommairement ce qu'est ODBC.

7.2 Implémentation d'un traducteur

7.2.1 Traducteur pour les requêtes accédant à une seule table

Le processus de traduction

Afin de bien comprendre notre démarche, la figure 7.2 illustre l'ensemble du processus de traduction que nous allons implémenter. Un paragraphe sera consacré à chacun des modules composant ce processus de traduction.

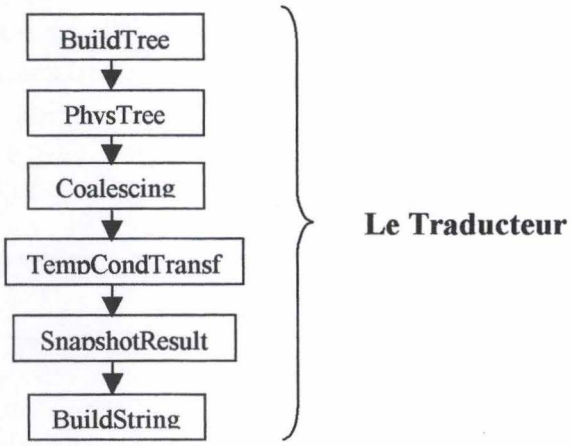


Figure 7.2: Schéma du traducteur et de ses différents modules.

Module 1: Représentation adéquate pour une requête mini-TSQL2

Avant d'établir les différentes règles de traduction, nous devons adopter une représentation adéquate pour manipuler les requêtes à traduire. Nous opterons pour une représentation sous forme d'arbre car elle offre une structure flexible qui se réorganise très facilement. Par exemple, nous pouvons aisément y ajouter ou supprimer un sous-arbre. La structure des cellules de l'arbre, que nous utiliserons, sera composée de trois champs, un pour y stocker le type de la cellule, un autre pour sa valeur et un troisième pour accéder à ses fils. L'annexe C détaille l'ensemble des types de cellules que nous utiliserons pour l'implémentation de notre traducteur. L'exemple de la figure 7.3 montre la correspondance existante entre une requête et l'arbre qui en est dérivé.

Pour réaliser cette étape de transformation d'une requête mini-TSQL2 en un arbre, un module appelé *Buildtree* sera implémenté.

```
select snapshot NOM,SALAIRE
from   PERSONNE
where  valid(PERSONNE)overlaps period '[10,15)',
       NOM='Dupuis'
```

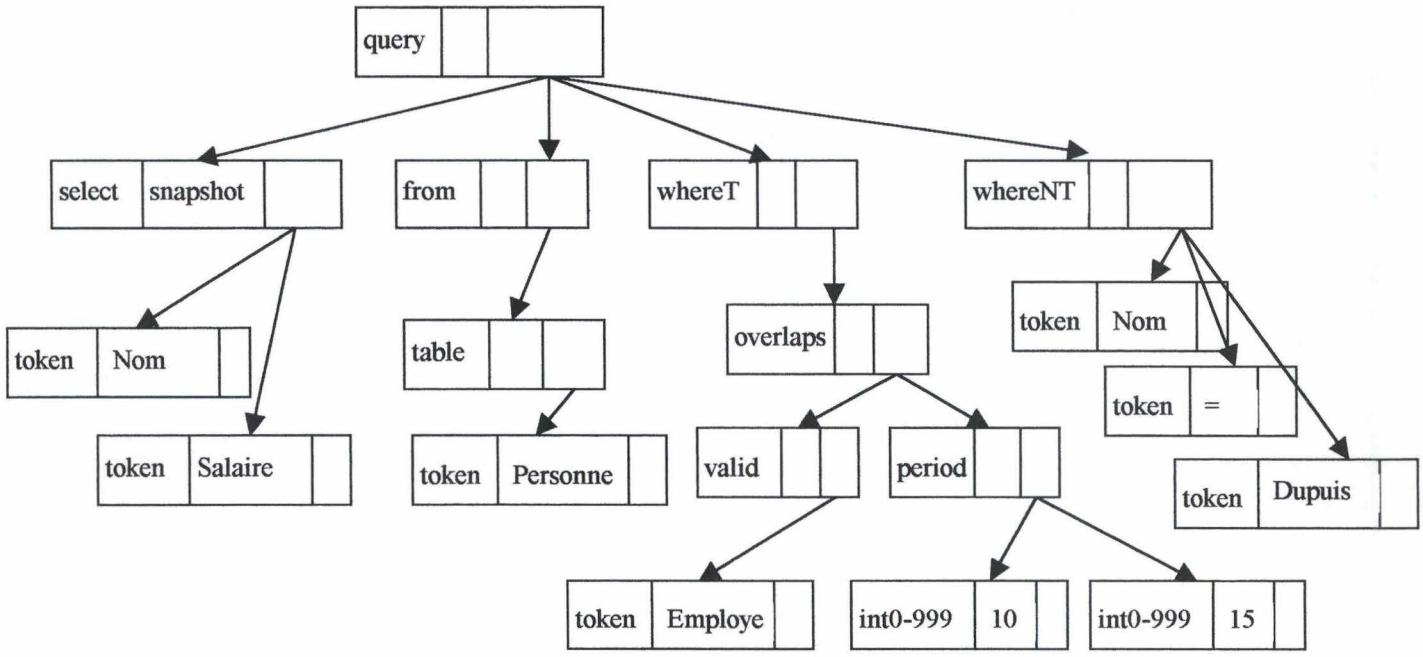


Figure 7.3: Exemple de relation entre une requête mini-TSQL2 et l'arbre de cette requête.

Module 2: Transformation d'une requête logique en requête physique

La seconde étape de notre traduction, consiste à transformer nos requêtes logiques (arbres) en requêtes dites physiques. En effet, comme nous l'avons déjà souligné au chapitre 6, les tables temporelles de la clause `from` de nos requêtes de sélection n'existent pas physiquement. Pour des raisons d'optimisations, certaines d'entre-elles ont dû être scindées en plusieurs tables. Pour pouvoir créer une requête physique, nous devons connaître la transformation subie par la table temporelle lors de la phase de pré-optimisation (chapitre 5). A cet effet, nous avons créé un dictionnaire temporel. Celui-ci aura pour but de nous renseigner sur la composition physique des bases de données temporelles. Il existera donc une bijection entre d'un côté, la table logique et le dictionnaire de données et de l'autre, la ou les tables physiques (figure 7.4).

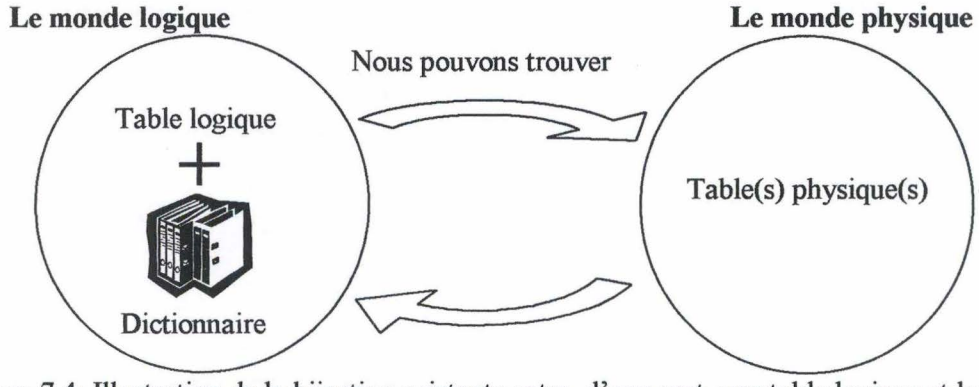


Figure 7.4: Illustration de la bijection existante entre, d'une part, une table logique et le dictionnaire et, de l'autre, la ou les tables physiques.

Pour implémenter le dictionnaire temporel, nous utiliserons une table normale. Cependant, pour rester cohérent avec les dictionnaires d'Interbase[1], le nom de celui-ci débutera par `RDB$`. Le dictionnaire temporel peut ne contenir que le numéro de la transformation qu'a subie la table logique temporelle ainsi que son type (MV, MT, B). En effet, dans ce cas et avec la notation des noms de tables temporelles physiques que nous connaissons (chapitre 5), le lien entre le nom de la table logique et celui (ceux) de la (des) table(s) physique(s) peut être établi. Cependant, si nous agissons de la sorte, nous contraignons l'utilisateur à adopter notre propre notation de tables physiques. C'est pour cette raison que les noms des tables physiques seront enregistrés dans le dictionnaire temporel, permettant ainsi à l'utilisateur de les changer.

Maintenant que nous connaissons le moyen d'obtenir l'ensemble des noms des tables physiques se rapportant à une table logique, nous devons déterminer le sous-ensemble optimal auquel notre requête doit accéder. Comme l'éclatement a été réalisé pour séparer les événements selon le temps qu'ils véhiculent, le temps sera un facteur déterminant. La matrice de la figure 7.5 montre l'ensemble optimal de tables physiques auxquelles nous devons accéder pour obtenir, selon le type de la table logique, la transformation subie et surtout par rapport au temps des données à extraire, les événements désirés.

Par exemple, si nous voulons obtenir les états courants (présents) d'une table logique bitemporelle nommée `PERSONNE` et qui a subi la transformation 2 , nous devrons accéder à la table physique `C_PERSONNE`.

Le problème mis en évidence par cette matrice concerne la transformation 2 dans le passé. En effet, dans ce cas bien précis l'union de deux tables physiques est requise pour accéder à l'ensemble des événements qui nous intéressent. Cependant, cette démarche semble contraire à la philosophie même de la transformation 2 qui préconisait une séparation des états courants et des états historiques pour mieux y accéder séparément. Or, dans notre cas, pour obtenir un historique complet, nous devons accéder aux données courantes. Ceci est dû au fait que l'état courant véhicule des informations sur le passé. Par exemple, si l'état courant de l'entité Léo est « Léo habite Namur depuis le 25 jusqu'à maintenant », l'information historique émanant de cet état est « Léo a habité à Namur du 25 jusqu'à hier ». Toutefois, cette transformation peut être judicieuse lorsque nous possédons énormément de données courantes car nous évitons la redondance de ces données dans la table historique. Pour ce cas particulier, un module devra être créé pour réaliser l'union des deux tables physiques. Comme ce module n'est pas difficile à implémenter nous considérerons dans nos réflexions que nous possédons déjà la jointure de ces différentes tables. Le nom des tables mono-temporelles et bitemporelles résultant de cette jointure seront préfixées respectivement par «C_U_H_» et «C_U_H_NV_» où «U» signifie union.

[1] Interbase est un système de gestion de bases de données relationnelles (SGBDR) qui fonctionne sur Windows 95, Windows NT, Novell, NetWare, et la plupart des implémentations de l'OS (« operating system») UNIX.

Table logique \ Temps	PRESENT	PAST	FUTURE
TRANSF: 1 **TABLE :** mono-temporelle valid time	C_F_H_...	C_F_H_...	C_F_H_...
TRANSF: 1 **TABLE:** mono-temporelle transaction time	C_H_...	C_H_...	
TRANSF: 1 **TABLE:** bitemporelle	C_F_H_NV_...	C_F_H_NV_...	C_F_H_NV_...
TRANSF: 2 **TABLE:** mono-temporelle valid time	C_...	C_... union H_...	
TRANSF: 2 **TABLE:** mono-temporelle transaction time	C_...	C_... union H_...	
TRANSF: 2 **TABLE:** bitemporelle	C_...	C_... union H_NV_...	
TRANSF: 3 **TABLE:** mono-temporelle valid time	C_...	C_H_...	
TRANSF: 3 **TABLE:** mono-temporelle transaction time	C_...	C_H_...	
TRANSF: 3 **TABLE:** bitemporelle	C_...	C_H_NV_...	

Figure 7.5: Ensemble optimal de tables physiques déterminé selon le type de la table logique, sa transformation et le temps .

Il reste à définir le moyen qui sera utilisé pour déterminer si une requête accède à des états passés, présents ou futurs. Nous savons que la transformation 1 est la seule à permettre le stockage d'information future et que celle-ci accède, quel que soit le temps, à la même table physique. Il est donc correct de dire que, quelle que soit la transformation, nous accéderons pour le passé ou le futur, si celui-ci existe, toujours à la même table physique. Notre problème est donc binaire. En effet, il s'agit de déterminer si notre requête accède ou n'accède pas aux états courants. Une requête accède aux états courants lorsqu'elle ne possède pas de condition temporelle ou lorsqu'elle possède uniquement une des conditions temporelles de la figure 7.6.

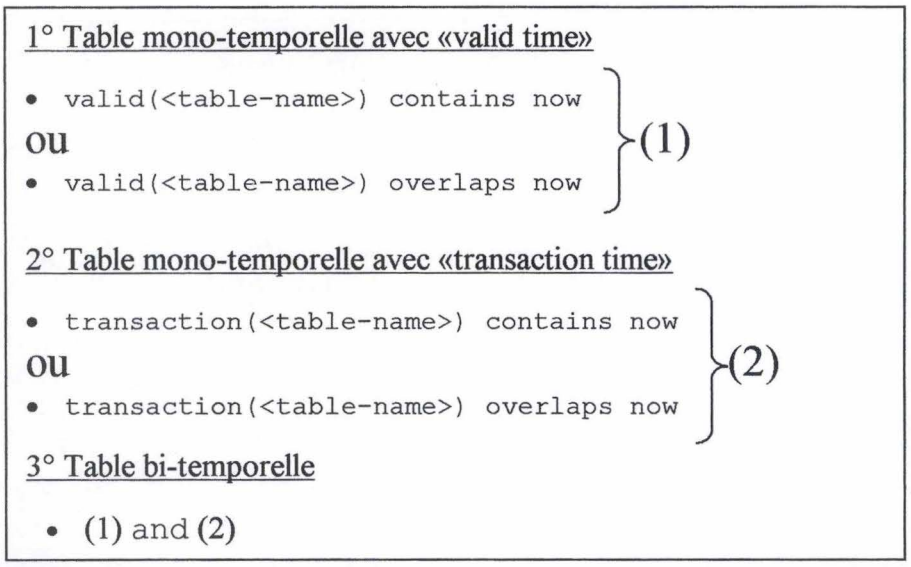


Figure 7.6: Conditions temporelles d'une requête accédant uniquement aux états courant.

Deux possibilités d'implémentation sont possibles. La première consiste en la création d'un module qui vérifiera, pour déterminer si une requête accède uniquement aux états courants, l'ensemble des conditions évoquées ci-dessus. Cependant, nous pouvons très bien nous limiter à la détection de l'absence de conditions temporelles. Dans ce cas, nous considèrerons que la présence d'une des conditions de la figure 7.6 est un abus de langage. En effet, par défaut une requête accède aux états courants. Dans la deuxième implémentation, une requête qui spécifie explicitement l'accès aux données courantes, sera considérée comme n'importe quelle requête passée ou future. Comme toutes les tables physiques passées et futures comportent aussi les données présentes, la requête fournira une solution correcte mais en un temps d'exécution plus long. Les avantages et inconvénients de ces deux implémentations de détection d'accession aux données courantes sont exposés à la figure 7.7.

Module complet de détection

Avantages:

- toutes les requêtes d'accès aux données courantes sont détectées.

Inconvénients:

- implémentation moyennement difficile.
- temps d'exécution du module plus important que dans le cas d'une détection partielle.

Module partiel de détection

Avantages:

- implémentation facile.
- temps d'exécution du module dérisoire.

Inconvénients:

- seules les requêtes sans conditions temporelles sont détectées.

Figure 7.7: Avantages et inconvénients des deux types de détection d'accèss aux données courantes.

En ce qui concerne le dictionnaire temporel, celui-ci enregistrera donc seulement deux noms des tables physiques : un pour les données présentes uniquement et un pour les données passées, présentes et futures. L'annexe D définit l'ensemble des données stockées par le dictionnaire temporel.

Hormis le cas de la transformation 2, pour passer d'une requête logique à une requête physique, le nom de la table logique dans l'arbre doit être remplacé par le nom de la table physique adéquate. Toutefois, cela ne suffit pas car le nom de la table logique se retrouve dans différentes parties de la requête. Par exemple, la table associée au « valid time » ou « transaction time », les alias, etc. Nous devons donc passer entièrement notre requête en revue pour changer l'ensemble des noms de la table logique.

Pour résumer, la deuxième étape de notre traducteur est composée de trois phases . La première doit rechercher les informations concernant la table logique dans le dictionnaire temporel en vue de déterminer l'ensemble des tables physiques correspondantes. Ensuite une seconde phase doit établir, grâce à un module de détection d'accès aux données courantes, la table physique optimale pour notre requête. Enfin, la troisième consiste à remplacer tous les noms de la table logique de l'arbre par le nom de la table physique adéquate. Le module qui implémente cette deuxième étape sera nommé *PhysTableTree*.

Module 3 : Résolution du coalescing

Le coalescing est réalisable grâce aux algorithmes vus au chapitre 4. Concrètement, notre implémentation doit, tout d'abord, déterminer l'identifiant d'entité de la table sur laquelle le coalescing va être effectué. Ceci est réalisable grâce à l'introduction d'un nouveau dictionnaire `RDB$TEMP_KEY` qui enregistre l'identifiant d'entité d'une table logique. Ensuite, notre module doit vérifier si l'identifiant d'entité est entièrement repris dans notre requête de sélection. Si ce n'est pas le cas, le coalescing sera réalisé comme si l'identifiant d'entité en faisait partie.

Un problème subsiste. En effet, l'algorithme de coalescing détaillé au chapitre 4 passe en revue les enregistrements de la table à synthétiser pour créer une table résultat. Celle-ci n'est rien d'autre qu'une table temporaire qui sera supprimée lorsque la sélection des enregistrements finals satisfaisant aux conditions temporelles et non temporelles aura été effectuée. Cependant, cette table existera physiquement dans notre base de données durant un court moment. Le problème rencontré réside dans le nom que pourrait porter cette table. En effet, nous devons être sûrs que celui-ci n'existe pas déjà dans notre base de données. La solution à ce problème est simple : soit l'utilisateur spécifie explicitement le nom de la table résultant du coalescing, soit celui-ci est généré automatiquement à l'aide de standards de notation. Dans les deux cas, de toutes façons, il existera autant de noms de tables temporaires que de tables à synthétiser dans la requête de sélection. Une fois le nom de la table résultat défini, nous devons être capables de la créer physiquement. A cet effet, nous utiliserons les dictionnaires de données existant dans tout SGDB qui décrivent la composition de chaque table physique.

En exposant uniquement au coalescing les données répondant aux conditions non temporelles, nous diminuons le nombre d'enregistrements stockés dans la table résultat. De plus, cette prise en compte des conditions non temporelles avant le coalescing améliore notre algorithme. Exactement comme décrit au chapitre 4, nous diminuons encore le temps de rotation des tables de travail du coalescing et le temps de passage d'un enregistrement d'une table de travail à l'autre.

Pour augmenter la vitesse d'exécution du coalescing, notre méthode va être améliorée. En effet, avant de procéder au coalescing, nous pouvons créer un petit module qui permettrait d'effectuer rapidement une vérification de la pertinence d'une demande de coalescing pour une requête déterminée. Ce module utilisera un nouveau dictionnaire temporel, appelé `RDB$TEMP_COLUMN`, qui enregistrera l'ensemble des données relatives aux colonnes d'une table logique. Ce module examinera si la requête demandeuse d'un coalescing sélectionne l'ensemble des colonnes temporelles. Si c'est le cas, le coalescing qui n'est pas utile ne sera pas effectué.

Comme pour la deuxième étape, les noms des tables physiques vont être remplacés par les noms des tables correspondantes résultant du coalescing. Le module qui traite du coalescing sera nommé *Coalescing*.

Module 4: Transformation des conditions temporelles

Cette étape a pour but d'effectuer la transformation des conditions temporelles, décrite dans notre langage mini-TSQL2, en conditions normales pour SQL. Cette transformation est très simple et se déroule en trois phases.

Lors de la première, tous les datetimes particuliers `now` sont remplacés par la valeur temporelle actuelle. En SQL, il existe un moyen pour connaître selon la granularité demandée la valeur du moment présent. Toutefois, comme nous travaillons avec des entiers, nous devons obligatoirement pallier l'absence de la connaissance du moment présent. Pour ce faire, la table `RDB$TEMP_NOW` sera utilisée. Nous y stockerons l'entier représentant le moment présent. Cette phase consistera donc à accéder à la table `RDB$TEMP_NOW`, à y extraire la valeur présente et à remplacer tous les datetimes `now` de notre requête par cette valeur. A la fin de cette phase, nous avons la certitude que tous les datetimes sont exprimés à l'aide d'entiers.

On transforme tous les timepoints sous forme de périodes. Comme nous l'avons souligné dans le chapitre 6, tous les timepoints peuvent être considérés comme des périodes. A la fin de cette deuxième phase, notre requête ne véhiculera que des informations temporelles sous forme de périodes.

Finalement, les opérateurs temporels entre deux périodes sont transformés selon les règles de la figure 7.8.

Mini-TSQL2: period'[i_1,i_2)' **PRECEDES** period'[i_3,i_4)' ⇒ *SQL*: $i_2 < i_3$

Mini-TSQL2: period'[i_1,i_2)' **EQUALS** period'[i_3,i_4)' ⇒ *SQL*: $(i_1 = i_3)$ **and** $(i_2 = i_4)$

Mini-TSQL2: period'[i_1,i_2)' **MEETS** period'[i_3,i_4)' ⇒ *SQL*: $i_2 = i_3$

Mini-TSQL2: period'[i_1,i_2)' **CONTAINS** period'[i_3,i_4)' ⇒ *SQL*: $(i_1 <= i_3)$ **and** $(i_2 >= i_4)$

Mini-TSQL2: period'[i_1,i_2)' **OVERLAPS** period'[i_3,i_4)' ⇒ *SQL*: $(i_2 > i_3)$ **and** $(i_1 < i_4)$

Mini-TSQL2: period'[i_1,i_2)' **STARTS** period'[i_3,i_4)' ⇒ *SQL*: $i_1 = i_3$

Mini-TSQL2: period'[i_1,i_2)' **FINISHES** period'[i_3,i_4)' ⇒ *SQL*: $i_2 = i_4$

Figure 7.8: Règles de transformations des opérateurs temporels de mini-TSQL2 en opérateurs standard d'SQL.

Lorsque cette troisième phase est terminée, nos conditions temporelles sont devenues des conditions normales SQL.

Le module permettant de réaliser l'ensemble de cette étape s'appelle *TempCondTransf*

Module 5 : Résolution du snapshot

Cette étape consiste à concrétiser la présence ou l'absence du littéral `snapshot`. De manière concrète, en l'absence de celui-ci, cette étape ajoutera les colonnes appropriées au type du résultat de la requête. Par exemple, si le résultat de la requête est de type mono-temporel avec «valid time», les colonnes `V_DEBUT` et `V_FIN` seront ajoutées à l'ensemble des colonnes de la requête. Ce module sera appelé *SnapshotResult.*

Module 6: Transformation de l'arbre en une requête SQL correcte

La sixième et dernière étape, nommée *BuildString*, parcourt et interprète l'arbre pour construire une requête SQL normale.

7.2.2 Traducteur pour les requêtes accédant à deux tables

Le problème de l'implémentation du langage mini-TSQL2 pour les requêtes accédant à deux tables réside dans la connaissance de l'existence et du sens de la clé étrangère. Supposons que ces informations se trouvent dans le dictionnaire temporel `RDB$TEMP_KEY`. Grâce à ces informations et à

celles relatives aux types de tables (RDB$TEMP_TABLE), nous pouvons déterminer avec précision dans quel cas de figure cette jointure doit être réalisée. L'étude que nous avons menée au chapitre 4 recense tous les cas de figures possibles entre deux tables et définit pour chacun de ceux-ci la traduction SQL de la condition de jointure normale et temporelle. La figure 7.18 résume la traduction effectuée dans chacun des cas de figure. Toutefois, certains cas ne seront pas abordés. En effet, nous ne traiterons pas des jointures temporelles dont la table référencée comporte moins d'informations, en terme temporel, que la table qui référence. Comme nous l'avons souligné au chapitre 4, ces cas particuliers demandent une traduction complexe.

Une fois de plus, pour une question de lisibilité, IMAX et IMIN ont été utilisés dans la figure 7.10. Pourtant, ceux-ci n'existent pas en SQL92. Un moyen, pour réaliser ces deux fonctions, doit donc être trouvé. Pour notre implémentation, IMAX et IMIN seront créés grâce à des « user-defined functions » (UDF). Une UDF est une fonction écrite dans un langage de programmation normal, par exemple le langage C, et existe dans une librairie en dehors de la base de données. Le lien entre une UDF et la base de données est réalisé à l'aide d'une déclaration de fonction externe. Celle-ci s'effectue dans la base de données elle-même et permet d'associer à un nom, par exemple IMAX ou IMIN, une UDF particulière par exemple *fn_imax* et *fn_imin*. La figure 7.9 nous donne un aperçu de l'implémentation aisée des fonctions *fn_imax* et *fn_imin* en langage C.

```
/*=============================================================
 fn_imax(a,b) - Returns the maximum of two arguments
=============================================================*/
long EXPORT fn_imax(ARG(long*, a), ARG(long*, b))
ARGLIST(long *a)
ARGLIST(long *b)
{
 return  (*a > *b) ? *a : *b;
}

/*=============================================================
 fn_imin(a,b) – Returns the minimum of two arguments
=============================================================*/
long EXPORT fn_imin(ARG(long*,a),ARG(long*,b))
ARGLIST(long *a)
ARGLIST(long *b)
{
 return  (*a < *b) ? *a : *b;
}
```

Figure 7.9: Implémentation des deux UDFs fn_max et fn_min en langage C.

Cas de figures TAB1 ⇒ TAB2	Condition de jointure mini-TSQL2	Traduction SQL de la condition de jointure	Type de l table résultat	Traduction SQL de la sélection des périodes
NT ⇒ NT	[TAB1.]COL1 == TAB2.]COL2	[TAB1.]COL1 = [TAB2.]COL2	NT	
NT ⇒ MV		[TAB1.]COL1 = [TAB2.]COL2 and V_FIN = 999	NT	
NT ⇒ MT		[TAB1.]COL1 = [TAB2.]COL2 and T_FIN = 999	NT	
NT ⇒ B		[TAB1.]COL1 = [TAB2.]COL2 and V_FIN = 999 and T_FIN = 999	NT	
MV ⇒ MV		[TAB1.]COL1 = [TAB2.]COL2 and TAB1.V_DEBUT < TAB2.V_FIN and TAB2.V_DEBUT < TAB1.V_FIN	MV	IMAX(TAB1.V_DEBUT, TAB2.V_DEBUT) IMIN(TAB1.V_FIN, TAB2.V_FIN)
MT ⇒ MT		[TAB1.]COL1 = [TAB2.]COL2 and TAB1.T_DEBUT < TAB2.T_FIN and TAB2.T_DEBUT < TAB1.T_FIN	MT	IMAX(TAB1.T_DEBUT, TAB2.T_DEBUT) IMIN(TAB1.T_FIN, TAB2.T_FIN)
MV ⇒ B		[TAB1.]COL1 = [TAB2.]COL2 and TAB1.V_DEBUT < TAB2.V_FIN and TAB2.V_DEBUT < TAB1.V_FIN and TAB2.T_FIN = 999	MV	IMAX(TAB1.V_DEBUT, TAB2.V_DEBUT) IMIN(TAB1.V_FIN, TAB2.V_FIN)
B ⇒ B		[TAB1.]COL1 = [TAB2.]COL2 and TAB1.V_DEBUT < TAB2.V_FIN and TAB2.V_DEBUT < TAB1.V_FIN and TAB1.T_DEBUT < TAB2.T_FIN and TAB2.T_DEBUT < TAB1.T_FIN	B	IMAX(TAB1.V_DEBUT, TAB2.V_DEBUT) IMIN(TAB1.V_FIN, TAB2.V_FIN) IMAX(TAB1.T_DEBUT, TAB2.T_DEBUT) IMIN(TAB1.T_FIN, TAB2.T_FIN)

Figure 7.10: Tableau de traduction des conditions de jointure temporelle et des sélections des périodes dans le résultat selon tous les cas de figure.

De manière concrète, deux modules vont devoir accroître leurs fonctionnalités : le module *TempCondTransf* et le module *SnapshotResult*. Le premier, en cas de requête sur deux tables, aura pour but de traduire la condition de jointure temporelle. Quant au deuxième, il devra ajouter ou non selon l'apparition du littéral `snapshot` et, selon le cas de figure, les colonnes supplémentaires représentant les différentes périodes. Par ailleurs, deux autres modules vont subir une légère modification. En effet, les modules *PhysTree* et *Coalescing* doivent maintenant s'exécuter sur les deux tables de la requête.

La figure 7.11 résume le cheminement de notre implémentation sur une requête générique lorsque le coalescing est effectué séparément sur les deux tables `TAB1` et `TAB2` de la requête.

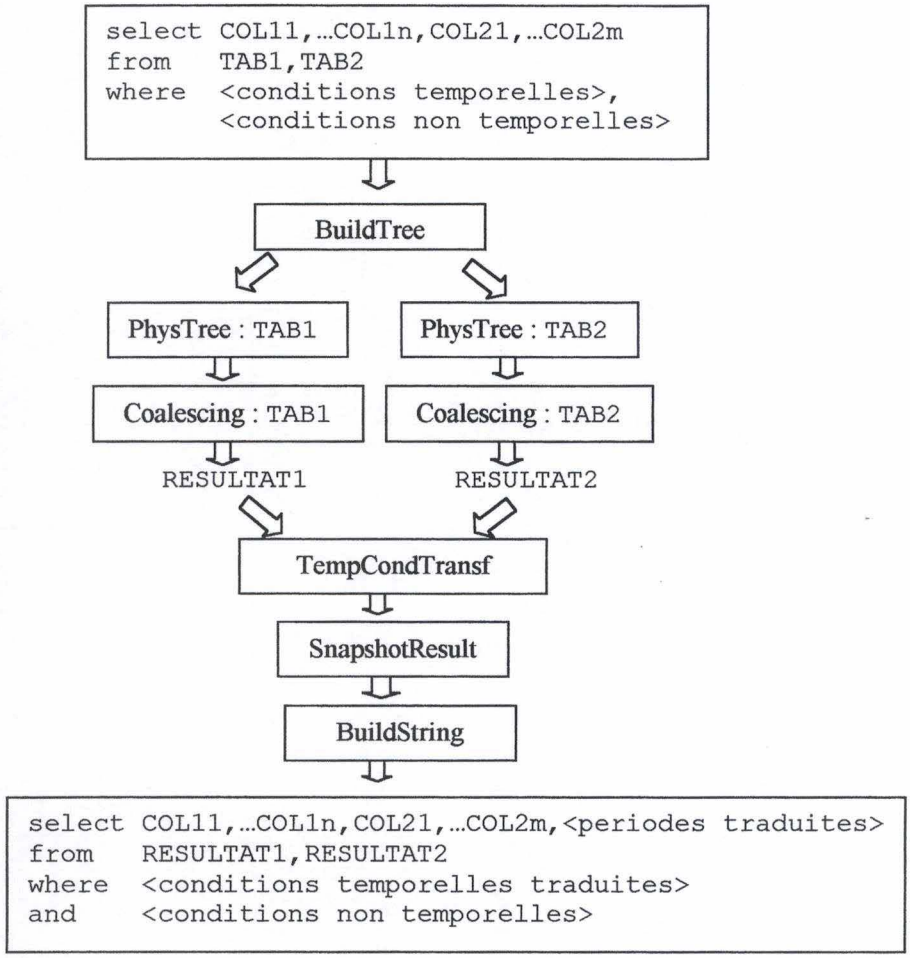


Figure 7.11 : Processus de traduction d'une requête générique accédant à deux tables lorsque le coalescing est réalisé avant la jointure temporelle.

7.3 Amélioration d'ODBC : TODBC

7.3.1 Pourquoi utiliser ODBC ?

Lors de la mise au point du langage mini-TSQL2, nous avons dû faire un choix d'implémentation. Celui-ci résidait dans le mode d'accès aux bases de données. En effet, en SQL, il existe 3 modes d'accès d'un programme à une base de données: le mode intégré (*embedded SQL*), le mode module (*SQL modular*) et le mode API SQL (Application Programming Interface). Avant de déterminer le mode que nous avons adopté, nous nous devons de développer le raisonnement qui nous a poussé à faire ce choix.

Dès le début de l'implémentation de notre langage, deux problèmes majeurs ont vu le jour. Le premier est lié spécifiquement à l'implémentation de notre traducteur de requêtes mini-TSQL2. En effet, à l'intérieur de celui-ci, nous devons être capables de réaliser des requêtes SQL dynamiques qui, par définition, ne sont connues qu'au moment de l'exécution et non pas à la compilation. Cela permet à notre traducteur, lors de chaque appel, de générer des requêtes internes différentes. Par exemple, dans notre traducteur, nous avons une requête dynamique qui accède à la table `RDB$TEMP_TABLE`. Cette requête, lors de l'exécution, est instanciée du nom de la table logique dont nous voulons récupérer les informations physiques. Cette première constatation nous pousse à choisir un mode d'accès aux bases de données qui soit le plus dynamique possible.

Le deuxième problème rencontré résidait dans l'exécution de la traduction de nos requêtes mini-TSQL2. En effet, une fois la traduction accomplie, un moyen d'exécuter dynamiquement cette chaîne de caractère doit être trouvé. Contrairement au premier problème, ici, l'entièreté de la requête n'est connue que lors de l'exécution.

Examinons chacun des modes d'accès en soulignant leurs apports éventuels à la résolution de nos problèmes.

Dans le mode intégré, les requêtes SQL sont placées directement dans le programme. Celui-ci est écrit dans un langage particulier appelé « langage hôte ». La construction de l'application passe par l'appel à un précompilateur qui se charge d'analyser et de remplacer chaque instruction SQL par une ou plusieurs instructions «équivalentes» en langage hôte [DELMAL, 98]. Le mode intégré ne permet pas de réaliser l'implémentation de notre langage mini-TSQL2 car il n'est pas assez dynamique. En effet, celui-ci ne permet pas de générer des requêtes qui varient d'un appel à l'autre par le nombre de ses éléments dynamiques. Pour expliquer plus précisément cette constatation, introduisons deux degrés de dynamisme parmi les requêtes. Le premier concerne les requêtes dynamiques dont la structure exacte est connue mais dont certains composants sont sans valeur. Les différentes valeurs seront fixées lors de l'exécution. La figure 7.12 donne un exemple de ce type de requête en mode intégré. *Eleve, cours* et *points_obtenu* sont les variables qui seront instanciées au moment de l'exécution.

```
....
EXEC SQL BEGIN DECLARE SECTION ;
char eleve[10], cours[10] ;
float points_obtenus ;
EXEC SQL END DECLARE SECTION ;

EXEC SQL insert into resultats
            values( :eleve, :cours, :points_obtenus) ;
....
```

Figure 7.12: Exemple de requête dynamique du premier degré en mode intégré.

Les requêtes dynamiques de second degré sont des requêtes dont la structure peut changer d'un appel à l'autre. Par exemple, une requête dynamique de création de table peut voir son nombre de colonnes changer lors de chaque appel. Un lien existe entre l'exécution d'une chaîne de caractère et la réalisation de requêtes dynamiques de second degré. En effet, s'il nous est possible de réaliser le premier, le second sera réalisable en considérant toute requête dynamique comme une chaîne de caractères. Puisque le mode intégré ne permet pas l'exécution de requêtes dynamiques de second degré que requiert notre implémentation, nous allons analyser les possibilités qu'offrent les autres modes d'accès.

Un module SQL est une unité compilable séparément composée d'un ensemble de commande SQL [DELMAL, 98]. Dans le mode module, il n'existe pas de phase de précompilation, car les modules SQL sont externes à l'application. Dans ce mode d'accès, l'application qui est écrite dans un langage particulier (langage hôte) est capable d'appeler les différents modules SQL externes. Ce type d'accès aux bases de données possède néanmoins exactement les mêmes inconvénients que le mode intégré.

Le mode d'accès API SQL ne fait aucune distinction entre requêtes statiques et dynamiques. En fait, toutes les requêtes sont traitées comme des requêtes dynamiques [DELMAL, 98]. Le mode d'accès API SQL permet l'exécution d'une chaîne de caractères représentant une requête. De plus, comme nous l'avons souligné précédemment, si le mode d'accès permet l'exécution d'une requête sous forme de chaîne de caractères alors il permet aussi la réalisation de requêtes dynamiques de second degré. Nous utiliserons donc pour notre implémentation ODBC (Open DataBase Connectivity) de Microsoft qui est la norme de fait en matière d'API SQL.

7.3.2 Présentation sommaire d'ODBC

ODBC est un produit de Microsoft qui a vu le jour en 1992. ODBC fonctionne sur une architecture client-serveur. Cela signifie que le programme (client) envoie ses requêtes à un SGBD (serveur) grâce à une API particulière. L'API d'ODBC est un ensemble de fonctions que n'importe quelle application peut utiliser. La caractéristique principale d'ODBC est de permettre une standardisation des requêtes. En effet, quelque soit le type de la base de données cible, une requête

ODBC s'écrira toujours de la même façon ; ODBC se chargeant de la traduire en une requête conforme au SGBD cible. Le fonctionnement d'ODBC est assez simple. Lorsqu'une application invoque une fonction de l'API d'ODBC, l'ensemble des informations de celle-ci est envoyé au gestionnaire de pilotes d'ODBC (*driver manager*). Les fonctions de l'API servent donc de liens entre l'application et le gestionnaire de pilotes. Celui-ci possède un rôle générique. Par exemple, il monte et démonte les pilotes spécifiques et gère les connections multiples. Son rôle principal est de servir d'interface générique pour les différents pilotes spécifiques. Ceux-ci ont pour but de dialoguer avec un SGDB particulier. La figure 7.13 illustre le fonctionnement d'ODBC.

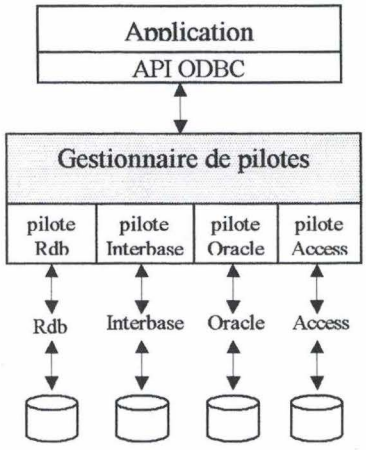


Figure 7.13: Principes de fonctionnement d'ODBC.

7.3.3 TODBC

L'implémentation de notre traducteur se fera donc à l'aide de fonctions ODBC. Toutefois, nous devons encore définir comment l'utilisateur va émettre une requête mini-TSQL2. Concrètement, ODBC va être amélioré de manière à pouvoir prendre en compte les instructions du langage mini-TSQL2. Nous appellerons TODBC (Temporal ODBC) cette nouvelle version d'ODBC. TODBC supportera donc non seulement SQL mais aussi notre langage mini-TSQL2. En d'autres termes, TODBC possèdera de nouvelles fonctions d'API qui manipuleront les bases de données temporelles. La figure 7.14 illustre notre démarche.

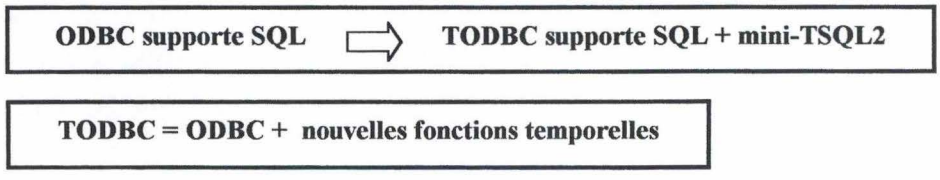


Figure 7.14: Relation entre ODBC et TODBC.

Les nouvelles fonctions que nous allons introduire doivent rester cohérentes avec les fonctions d'ODBC existantes. C'est pour cette raison que nous allons étudier comment une requête SQL est réalisée en ODBC. Nous pourrons ensuite calquer cette étude pour une requête mini-TSQL2. La figure 7.15 nous donne l'architecture générale d'un programme ODBC lorsque nous utilisons ODBC 3.5. Tout programme ODBC commence par initialiser l'interface ODBC en

allouant un descripteur d'environnement et de connexion. Pour plus de précisions sur ces deux concepts, le lecteur est invité à lire [DELMAL, 98]. Lorsque la phase d'initialisation est réalisée, il est possible d'établir la ou les connexions souhaitée(s) sur la ou les base(s) de données cible(s). Une fois connecté, le client peut demander le traitement de requêtes. A la fin du programme, le client doit se déconnecter et libérer les ressources utilisées pour le descripteur de connexion et d'environnement.

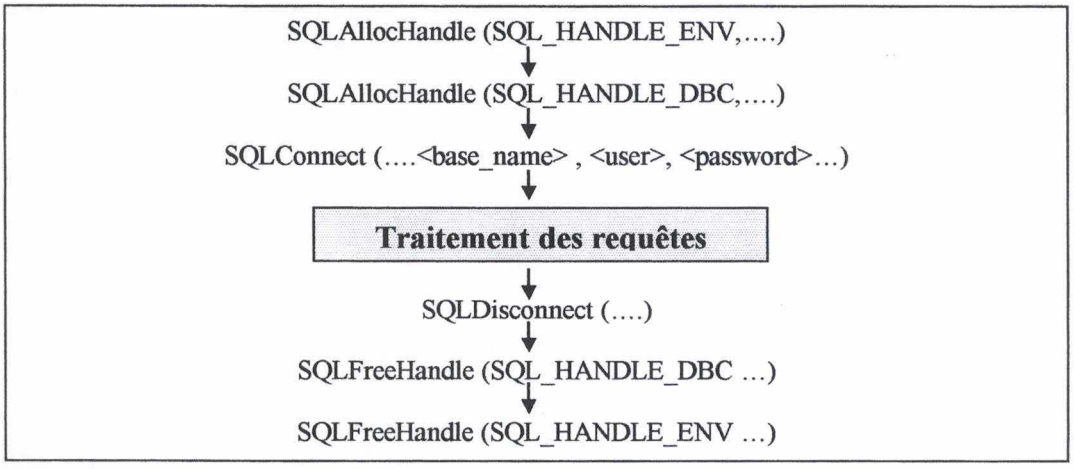


Figure 7.15: Architecture d'un programme ODBC.

Quel que soit le langage dans lequel vont être réalisées les requêtes (SQL ou mini-TSQL2), cette structure de programme restera toujours la même. La différence va donc s'opérer lors du traitement des requêtes. En ODBC comme en TODBC, le traitement des requêtes de sélection SQL se déroule toujours de la même façon. Un descripteur de requête est d'abord alloué. Ensuite, la requête est exécutée au moyen d'un appel à SQLExecDirect. Puis, la récupération des résultats se fait à l'aide de SQLBindCol et de SQLFetch. SQLBind permet de lier une variable du programme à une colonne de la sélection tandis que SQLFetch permet d'extraire un tuple de l'ensemble résultat. Une fois tous les résultats récupérés, le descripteur de requête peut être désalloué. L'ensemble de ce processus est expliqué de manière précise dans [DELMAL,98].

Pour rester cohérent, les nouvelles fonctions créées dans TODBC, pour notre langage mini-TSQL2, suivront le même déroulement pour réaliser une requête de sélection mini-TSQL2. Notre implémentation se limite à la création de fonctions intrinsèquement nouvelles. C'est ainsi que SQLBindCol et SQLFetch garde la même sémantique lors de la réalisation d'une requête mini-TSQL2. La figure 7.16 montre la ressemblance existante entre le traitement d'une requête de sélection SQL et mini-TSQL2 en TODBC.

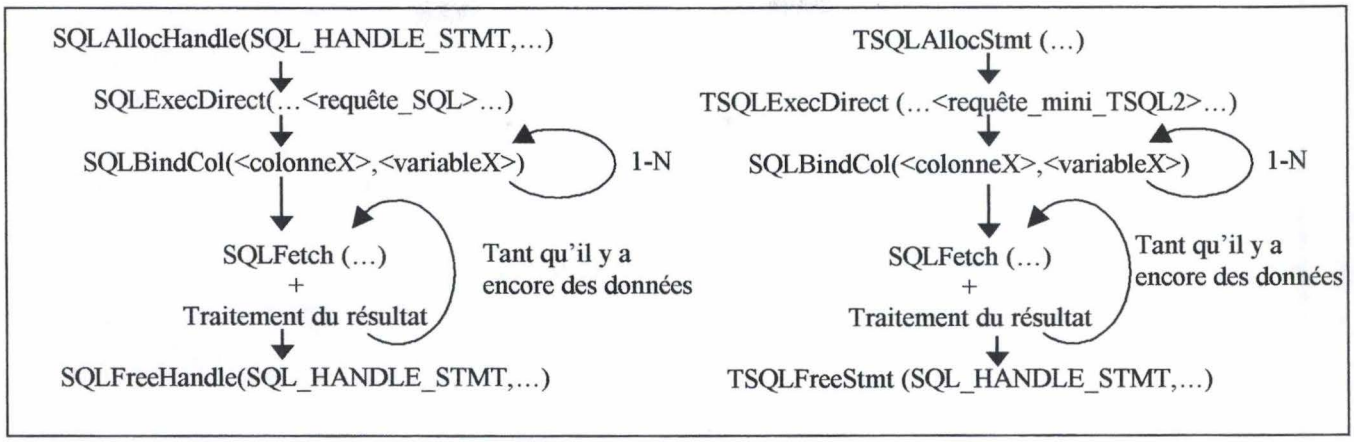


Figure 7.16: Traitement d'une requête SQL et mini-TSQL2 en TODBC.

Toutes les nouvelles fonctions de TODBC élaborées pour traiter les requêtes mini-TSQL2 sont préfixées par la chaîne de caractères « TSQL ». Pour les requêtes de sélection, trois fonctions sont utilisées : TSQLAllocStmt, son opposé TSQLFreeStmt et la plus importante TSQLExecDirect. Détaillons chacune de ces fonctions.

TSQLExecDirect a le même rôle que la fonction SQLExecDirect. Ce qui les différencie, c'est le langage de la requête à exécuter. En effet, TSQLExecDirect est utilisé pour préparer et exécuter les requêtes mini-TSQL2 tandis que pour les requêtes SQL, l'utilisation d'SQLExecDirect est requis. Les figures 7.17 et 7.18 nous donnent l'interface de ces deux fonctions.

Syntaxe:
RETCODE **SQLExecDirect** (HSTMT StatementHandle, SQLCHAR * StatementText,
SQLINTEGER TextLength);
Arguments:
StatementHandle [Input] : le descripteur de requête.
StatementText [Input] : la requête SQL a exécuter.

Figure 7.17 Interface de la fonction SQLExecDirect.

Syntaxe:
RETCODE **TSQLExecDirect** (HDBC ConnexionHandle, HSTMT StatementHandle,
SQLCHAR * StatementText);
Arguments:
ConnexionHandle [Input] : le descripteur de connexion
StatementHandle [Input] : le descripteur de requête.
StatementText [Input] : la requête mini-TSQL2 a exécuter.

Figure 7.18: Interface de la fonction TSQLExecDirect.

L'implémentation de TSQLExecDirect et de toutes les fonctions temporelles de TODBC se base sur le même principe. Celui-ci consiste à considérer nos nouvelles fonctions comme des méta-fonctions ODBC. En effet, chacune de ces fonctions appellera en son sein une ou plusieurs fonctions ODBC. Comme certaines fonctions ODBC encapsulées à l'intérieur d'une fonction temporelle ont besoin d'un descripteur de connexion, celui-ci doit être passé en paramètre. C'est le cas de TSQLExecDirect. Ce type d'implémentation est réalisable car en ODBC les descripteurs sont des pointeurs vers une zone de mémoire. Cela signifie que grâce au passage en paramètre de ces pointeurs, l'accession depuis une fonction temporelle à une zone de mémoire allouée dans un programme TODBC est possible.

Grâce à notre traducteur (paragraphe 7.2), l'implémentation de la fonction temporelle TSQLExecDirect est aisée. En effet, TSQLExecDirect va simplement fournir la requête mini-TSQL2 au traducteur qui génèrera une requête SQL équivalente. Ensuite, TSQLExecDirect n'aura plus qu'à exécuter la requête traduite à l'aide de la fonction ODBC SQLExecDirect. La figure 7.19 illustre cette implémentation.

```
TSQLExecDirect (...<requête_mini_TSQL2>...)
 {
  Traducteur(...<requête_mini_TSQL2>, <requête_SQL>...)
                          INPUT              OUTPUT
  SQLExecDirect (..<requête_SQL>...)
 } ;
```

Figure 7.19 : Implémentation de TSQLExecDirect.

Il reste néanmoins un problème à résoudre. Les requêtes SQL provenant de la traduction des requêtes mini-TSQL2 réalisent leur sélection sur une ou plusieurs table(s) temporaire(s). En effet, ces tables résultent de la phase de coalescing du processus de traduction. Il faut donc ne pas oublier de les supprimer lorsque tous les résultats auront été récupérés. C'est ce que va permettre la fonction TSQLFreeStmt. Pour implémenter TSQLFreeStmt, un tableau dynamique global aux deux fonctions TSQLExecDirect et TSQLFreeStmt devra être introduit. Ce tableau servira à stocker le nom de chaque table temporaire créée. Grâce à ce tableau, TSQLFreeStmt n'aura aucune difficulté à supprimer l'ensemble des tables temporaires. Pour rester cohérent dans notre approche, nous avons généré la fonction temporelle opposée à TSQLFreeStmt, TSQLAllocStmt. Celle-ci est l'image parfaite de son homologue ODBC puisqu'elle ne réalise qu'un simple appel à la fonction SQLAllocHandle. Les figure 7.20 et 7.21 donnent un aperçu de la syntaxe et de l'implémentation des deux fonctions.

Syntaxe:
RETCODE **TSQLAllocStmt**(HDBC ConnexionHandle, void * * StatementHandle);

Arguments:
ConnexionHandle [Input] : le descripteur de connexion
StatementHandle [Input] : le descripteur de requête.

Implémentation :

```
TSQLAllocStmt (...)
{
 SQLAllocHandle (SQL_HANDLE_STMT, ...)
} ;
```

Figure 7.20: Syntaxe et implémentation de TSQLAllocStmt.

Syntaxe:
RETCODE **TSQLFreeStmt**(HDBC ConnexionHandle, HSTMT StatementHandle);

Arguments:
ConnexionHandle [Input] : le descripteur de connexion
StatementHandle [Input] : le descripteur de requête.

Implémentation :

```
TSQLFreeHandle (...)
{
 Elimination des tables temporaires
 SQLFreeHandle (SQL_HANDLE_STMT, ...)
} ;
```

Figure 7.21: Syntaxe et implémentation de TSQLFreeStmt.

Pour être complet dans notre processus de traitement de requêtes de sélection, la fonction temporelle TSQLError doit être introduite. Celle-ci jouera le même rôle que la fonction SQLError pour ODBC1.0 et 2.x et que SQLGetDiagRec pour ODBC3.0. Ces différentes fonctions servent à récupérer et à diagnostiquer la cause d'un échec d'une fonction ODBC. C'est ce que va faire TSQLError pour les fonctions temporelles. L'implémentation de cette fonction utilisera un fichier texte. Ce fichier texte contiendra le descriptif de l'erreur rencontrée lors de l'exécution de nos fonctions temporelles. TSQLError devra donc accéder à ce fichier pour récupérer le message d'erreur désiré. Cette implémentation est très souple. En effet, elle permet de communiquer l'erreur d'une fonction ODBC interne à une fonction temporelle de TODBC. Plus précisément, la fonction temporelle utilisera, en cas d'erreur d'une fonction ODBC interne, SQLGetDiagRec et enregistrera dans notre fichier texte le descriptif de l'erreur. De plus, grâce à ce système, de nouveaux types

d'erreurs peuvent être créés. Il suffit pour cela d'écrire dans le fichier texte le type d'erreur désiré. La similitude de la gestion des erreurs entre une fonction normale et une fonction temporelle est illustrée par la figure 7.22.

```
RETCODE rc ;

rc = SQLExecDirect(...<requête_SQL>...) ;

if (rc == SQL_ERROR )
{
 SQLGetDiagRec (...) ; OU SQLError(...) ;
 ...
 « afficher le diagnostic de l'erreur »
 ...
}
```

```
RETCODE rc ;

rc = TSQLExecDirect(...<requête_mini_TSQL2>...) ;

if (rc == SQL_ERROR )
{
 TSQLError (...) ;
 ...
 « afficher le diagnostic de l'erreur »
 ...
}
```

Figure 7.22: Comparaison entre la gestion des erreurs pour une fonction normale et une fonction temporelle de TODBC.

La syntaxe et l'implémentation de la fonction temporelle TSQLError sont décrites à la figure 7.23.

Syntaxe
RETCODE TSQLError (SQLCHAR * Sqlstate, SQLCHAR * MessageText);

Arguments
Sqlstate [Output] :
Pointeur vers un buffer dans lequel se trouve un code SQLSTATE de 5 caractères. Les deux premiers caractères indiquent la classe, et les trois suivants indiquent la sous-classe. Les SQLSTATEs fournissent des informations détaillées sur la cause d'un warning ou d'une erreur.

MessageText [Output]
Pointeur vers un buffer dans lequel se trouve le message texte du diagnostic de l'erreur.

Implémentation

```
TSQLError (...)
{
 Récupération du diagnostic de l'erreur dans un fichier texte.
}
```

Figure 7.23: Syntaxe et implémentation de TSQLError.

Concrètement, comme pour ODBC, les nouvelles fonctions de TODBC seront implémentées dans une librairie. Celle-ci pourra donc être inclue par l'utilisateur dans tout programme et permettra l'appel à une ou plusieurs fonctions temporelles. Afin d'illustrer les capacités de TODBC, un programme permettant de réaliser une requête mini-TSQL2 de manière interactive sera implémenté. Cette application dénommée **W**indows **I**nteractive **TSQL** est exposée à l'annexe E.

Le choix d'implémenter l'exécution d'une requête mini-TSQL2 à l'aide de nouvelles fonctions temporelles nous offre d'énormes possibilités de flexibilité dans notre implémentation. Cela nous permet de trouver l'implémentation qui optimisera ou améliorera au mieux notre langage. Par exemple, certains concepts de notre langage mini-TSQL2 pourraient être facilités. C'est le cas, entre autres, du coalescing et de la possibilité d'incorporer ou non dans le résultat les différentes périodes. En effet, nous pourrions très bien simplifier notre langage en éliminant les littéraux `every` et `snapshot`. Ceux-ci seraient remplacés par deux nouveaux arguments dans notre fonction TSQLExecDirect. Ces arguments seraient de type booléen et permettraient de signaler respectivement notre désir de réaliser le coalescing ou d'incorporer les différentes périodes dans le résultat. Pour conserver la sémantique par défaut de nos requêtes mini-TSQL2, ces deux arguments possèderaient des valeurs par défaut. Il s'agirait du coalescing et de l'ajout des périodes dans le résultat. Cette nouvelle possibilité qu'offre les fonctions temporelles d'ajouter de nouveaux arguments intéressants, pourrait permettre aussi d'améliorer les performances de notre traducteur. Par exemple, le nombre de colonnes que la requête désire sélectionner pourrait être demandé en argument à la fonction TSQLExecDirect. Cela éviterait au traducteur de le calculer. Dans le même cadre, nous pourrions demander au programmeur de nous donner les noms des tables temporaires que le traducteur utiliserait pour le coalescing. Cela permettrait d'éviter tout problème de nom dans la base de données.

D'autres possibilités nous sont offertes par l'utilisation d'ODBC. Le lien (SQLBindCol) qui s'opère après l'exécution de la requête de sélection en est un exemple. En effet, nous pouvons très bien réaliser toutes les requêtes mini-TSQL2 sans l'utilisation du littéral ou de l'argument `snapshot`. Dans ce cas, toutes les requêtes de sélection incorporeront dans le résultat les périodes adéquates. Cependant, celles-ci peuvent n'être liées (SQLBindCol) à aucune variable dans le programme, ce qui reviendrait à l'utilisation du littéral ou de l'argument `snapshot`.

Une partie des considérations et des améliorations qui viennent d'être évoquées ont été implémentées dans l'application WITSQL de l'annexe E.

7.3.4 Exemple de programme TODBC

Ce programme réalise la requête de sélection mini-TSQL2 suivante : « `Select nom,salaire from Personne where valid(Personne) overlaps period'[10,20)'` ». Afin de gérer les éventuelles erreurs d'exécution, deux fonctions de détection des erreurs ont été créées : `detecter_erreur_SQL` et `detecter_erreur_TSQL`.

Détection des erreurs des fonctions normales de TODBC

```
void detecter_erreur_SQL(SWORD fHandleType, SQLHANDLE handle, char* comd)
{
 SQLINTEGER  fNative;
 SQLSMALLINT cb, psMsgNum;
 SQLCHAR     *szMessage;
 SQLCHAR     *szSQLState;
 RETCODE     rc;

 szMessage  = (SQLCHAR *) malloc(SQL_MAX_MESSAGE_LENGTH+1);
 szSQLState = (SQLCHAR *) malloc(6);
 psMsgNum   = 1;

 // extraction du message d'erreur

 rc=SQLGetDiagRec (fHandleType, handle, psMsgNum, szSQLState, &fNative,
                   szMessage,
                SQL_MAX_MESSAGE_LENGTH-1, &cb);

 if (rc == SQL_SUCCESS)
 {
  MessageBox(NULL,szMessage,comd,MB_OK);
  MessageBox(NULL,szSQLState, "SQLSTATE",MB_OK);
 };

 free(szMessage);
 free(szSQLState);
};
```

Détection des erreurs des nouvelles fonctions de TODBC

```
void detecter_erreur_TSQL (char* comd)
{
 SQLCHAR     *szMessage;
 SQLCHAR     *szSQLState;
 RETCODE     rc;

 szMessage  = (SQLCHAR *) malloc(SQL_MAX_MESSAGE_LENGTH+1);
 szSQLState = (SQLCHAR *) malloc(6);

 // extraction du message d'erreur

 rc= TSQLError(szSQLState, szMessage);

 if (rc == SQL_SUCCESS)
 {
  MessageBox(NULL,szMessage,comd,MB_OK);
  MessageBox(NULL,szSQLState, "SQLSTATE",MB_OK);
 };

 free(szMessage);
 free(szSQLState);
};
```

Gestion classique d'une sélection mini-TSQL2 : TODBC 3.0

```
#include <owl\pch.h>
#include <owl\applicat.h>
#include <owl\framewin.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include "wtsql.h"


//include for ODBC
# include <windows.h>
# include <sql.h>
# include <sqlext.h>


//include for TODBC
# include <todbc.h>


void detecter_erreur_SQL(SWORD fHandleType, SQLHANDLE handle, char* comd);
void detecter_erreur_TSQL (char* comd);


HENV henv;
HDBC hdbc;
HSTMT hstmt;
RETCODE rc;


class TMyWindow : public TWindow
{
 private :
 public :
 TMyWindow(TWindow *Parent =0);
 virtual void SetupWindow();
};

TMyWindow::TMyWindow(TWindow *Parent)
{
 Init(Parent,0,0);
}

void TMyWindow::SetupWindow()
{
 // Déclaration des variables qui vont recevoir les résultats

 char nom[50], salaire [50], output[110];
 SDWORD cbnom, cbsalaire ;

 // Connexion

 rc=SQLAllocHandle(SQL_HANDLE_ENV,SQL_NULL_HANDLE,&henv);
   if (rc==SQL_ERROR || rc==SQL_SUCCESS_WITH_INFO)
    {
     exit(1);
    };
```

```
rc=SQLSetEnvAttr(henv,SQL_ATTR_ODBC_VERSION,(void *)SQL_OV_ODBC2,0);
  if (rc==SQL_ERROR || rc==SQL_SUCCESS_WITH_INFO)
    {
     detecter_erreur_SQL(SQL_HANDLE_ENV,henv,"SQLSetEnvAttr");
     SQLFreeHandle(SQL_HANDLE_ENV,henv);
     exit(1);
    };

rc=SQLAllocHandle(SQL_HANDLE_DBC,henv,&hdbc);
  if (rc==SQL_ERROR || rc==SQL_SUCCESS_WITH_INFO)
    {
     detecter_erreur_SQL(SQL_HANDLE_ENV,henv,"SQLAllocHandle");
     SQLFreeHandle(SQL_HANDLE_ENV,henv);
     exit(1);
    };

rc=SQLSetConnectOption(hdbc,SQL_ATTR_ODBC_CURSORS,SQL_CUR_USE_ODBC);
  if (rc==SQL_ERROR || rc==SQL_SUCCESS_WITH_INFO)
    {
     detecter_erreur_SQL(SQL_HANDLE_DBC,hdbc,"SQLSetConnectOption");
     SQLFreeHandle(SQL_HANDLE_DBC,hdbc);
     SQLFreeHandle(SQL_HANDLE_ENV,henv);
     exit(1);
    };

rc=SQLConnect(hdbc,(UCHAR FAR *) "Mysource" , SQL_NTS,(UCHAR FAR *) "Sysdba",
              SQL_NTS,(UCHAR FAR *) "masterkey" , SQL_NTS);
  if (rc==SQL_ERROR || rc==SQL_SUCCESS_WITH_INFO)
    {
     detecter_erreur_SQL(SQL_HANDLE_DBC,hdbc,"SQLConnect");
     SQLFreeHandle(SQL_HANDLE_DBC,hdbc);
     SQLFreeHandle(SQL_HANDLE_ENV,henv);
     exit(1);
    }

// requête de sélection mini-TSQL2

rc=TSQLAllocStmt(hdbc,&hstmt);
  if (rc==SQL_ERROR || rc==SQL_SUCCESS_WITH_INFO)
    {
     detecter_erreur_TSQL("TSQLAllocStmt");
     SQLDisconnect(hdbc);
     SQLFreeHandle(SQL_HANDLE_DBC,hdbc);
     SQLFreeHandle(SQL_HANDLE_ENV,henv);
     exit(1);
    };

rc=TSQLExecDirect(hdbc,hstmt,"Select snapshot nom, salaire from Personne where
                valid(Personne) overlaps period'[10,20)'",type);
  if (rc==SQL_ERROR || rc==SQL_SUCCESS_WITH_INFO)
    {
     detecter_erreur_TSQL("TSQLExecDirect");
     TSQLFreeStmt(hdbc,hstmt);
     SQLDisconnect(hdbc);
     SQLFreeHandle(SQL_HANDLE_DBC,hdbc);
     SQLFreeHandle(SQL_HANDLE_ENV,henv);
     return ;
    };
```

```
rc=SQLBindCol(hstmt,1,SQL_C_CHAR,nom,50,cbnom);
if (rc==SQL_ERROR || rc==SQL_SUCCESS_WITH_INFO)
    {
     detecter_erreur_SQL(SQL_HANDLE_STMT,hstmt,"SQLBindCol");
     TSQLFreeStmt(hdbc,hstmt);
     SQLDisconnect(hdbc);
     SQLFreeHandle(SQL_HANDLE_DBC,hdbc);
     SQLFreeHandle(SQL_HANDLE_ENV,henv);
     exit(1);
    };

rc=SQLBindCol(hstmt,2,SQL_C_CHAR,salaire,50,cbsalaire);
if (rc==SQL_ERROR || rc==SQL_SUCCESS_WITH_INFO)
    {
     detecter_erreur_SQL(SQL_HANDLE_STMT,hstmt,"SQLBindCol");
     TSQLFreeStmt(hdbc,hstmt);
     SQLDisconnect(hdbc);
     SQLFreeHandle(SQL_HANDLE_DBC,hdbc);
     SQLFreeHandle(SQL_HANDLE_ENV,henv);
     exit(1);
    };
do
{
 rc = SQLFetch(hstmt);
 if(rc == SQL_NO_DATA)  break;
 if(rc==SQL_ERROR || rc==SQL_SUCCESS_WITH_INFO)
  {
   detecter_erreur_SQL(SQL_HANDLE_STMT,hstmt,"SQLFetch");
   break;
  };

 strcpy(output,"") ;
 strcat(output,nom);
 strcat(output," ");
 strcat(output,salaire);
 MessageBox(output,"TUPLE",MB_OK);

}while(rc!= SQL_NO_DATA);

// Déconnexion

rc=TSQLFreeStmt(hdbc,hstmt);
if (rc==SQL_ERROR || rc==SQL_SUCCESS_WITH_INFO)
    {
     detecter_erreur_TSQL("TSQLFreeStmt");
     SQLDisconnect(hdbc);
     SQLFreeHandle(SQL_HANDLE_DBC,hdbc);
     SQLFreeHandle(SQL_HANDLE_ENV,henv);
     exit(1);
    };

rc=SQLDisconnect(hdbc);
if (rc==SQL_ERROR || rc==SQL_SUCCESS_WITH_INFO)
    {
     detecter_erreur_SQL(SQL_HANDLE_DBC,hdbc,"SQLDisconnect");
     SQLFreeHandle(SQL_HANDLE_DBC,hdbc);
     SQLFreeHandle(SQL_HANDLE_ENV,henv);
     exit(1);
    };
```

```
rc=SQLFreeHandle(SQL_HANDLE_DBC,hdbc);
if (rc==SQL_ERROR || rc==SQL_SUCCESS_WITH_INFO)
    {
     detecter_erreur_SQL(SQL_HANDLE_ENV,henv,"SQLFreeHandle");
     SQLFreeHandle(SQL_HANDLE_ENV,henv);
     exit(1);
    };

rc=SQLFreeHandle(SQL_HANDLE_ENV,henv);
if (rc==SQL_ERROR || rc==SQL_SUCCESS_WITH_INFO)
    {
     exit(1);
    };

Twindow ::SetupWindow() ;

}

class Tapp : public Tapplication
{
public :
 Tapp() : Tapplication() {}
 void InitMainWindow()
 {
  SetMainWindow(new TframeWindow(0, »ODBC », new TmyWindow)) ;
 }
};

int OwlMain(int, char *[])
{
 return Tapp().Run() ;
}
```

8
Conclusions

8.1 Particularités du langage développé

Notre langage ainsi que son implémentation viennent s'ajouter aux travaux menés par de nombreux chercheurs. Nous sommes donc en droit de nous interroger sur l'opportunité et l'intérêt de notre travail. C'est pourquoi, nous voudrions à présent souligner les traits particuliers qui caractérisent celui-ci:

- ce travail se base sur des hypothèses bien définies issues du projet TimeStamp (chapitre 3); celles-ci forment un environnement de travail particulier,

- ce travail a étudié et a implémenté un opérateur de coalescing sur les différentes bases de données temporelles,

- ce travail a étudié et a implémenté un opérateur de jointure temporelle,

- ce travail a créé le langage mini-TSQL2 possédant une syntaxe simple et étant une extension intuitive du langage SQL,

- ce travail a comparé la puissance du langage mini-TSQL2 au langage TSQL2,

- ce travail a souligné la difficulté d'interprétation d'une requête mini-TSQL2 accédant à deux tables,

- ce travail a mis en évidence les nombreux choix sémantiques auxquels un constructeur de langages temporels est confronté,

- ce travail a démontré que la jointure de deux tables coalescées équivaut au coalescing de la jointure de deux tables,

- ce travail a tenu compte, lors de l'implémentation, de l'existence d'une phase de pré-optimisation,

- ce travail a illustré l'existence d'un pont entre notre langage et celui d'SQL,

- ce travail a mis en évidence la possibilité d'étendre les capacités d'ODBC grâce à de nouvelles fonctions temporelles.

Par ailleurs, au cours de l'implémentation de notre langage, nous nous sommes toujours efforcés de trouver un équilibre entre performance, lisibilité de notre langage de requêtes et facilité d'emploi des nouvelles fonctions temporelles de TODBC.

Soulignons enfin que notre recherche menée sur les bases de données temporelles n'a pas pour but de déterminer un langage idéale mais bien d'éveiller le lecteur aux problèmes rencontrés lors de la création et de l'implémentation d'un langage temporel.

8.2 Perspectives d'avenir

La recherche que nous avons menée est une première approche dans le cadre du projet TimeStamp. Il convient à l'avenir d'améliorer notre langage ainsi que son implémentation.

La première perspective d'amélioration serait d'étendre notre langage mini-TSQL2 à l'ensemble des requêtes temporelles possibles. Les requêtes de création et de suppression de tables temporelles ainsi que les requêtes d'insertion, de suppression et de mise à jour des données pourraient étendre les capacités de notre langage.

Pour l'implémentation de notre langage, nous pourrions également imaginer l'extension des fonctions temporelles de TODBC. Celles-ci permettraient d'augmenter les services offerts aux programmeurs dans le cadre d'une application. Par exemple, des fonctions permettant de récupérer les méta-données de nos bases temporelles pourraient être créées. De plus, une étude approfondie sur l'optimisation des requêtes mini-TSQL2 pourrait être réalisée en vue d'augmenter la rapidité des réponses des fonctions temporelles de TODBC.

La représentation du temps dans les bases de données doit pouvoir rencontrer les besoins du 21éme siècle. Un certain nombre de tentatives dont notre travail fait partie ont été effectuées pour y parvenir mais jusqu'à présent aucune approche commune n'a été élaborée. Cependant, la communauté de chercheurs en bases de données temporelles tend à renverser cette situation. Il convient, dans le futur, de suivre l'évolution de ce domaine de recherche et de travailler dans son sens en vue de l'élaboration de standards internationaux.

Bibliographie

[DELMAL, 98], DELMAL P., *SQL2 Application à Oracle, Access et RDB*, De Boeck & Larcier s.a., 1998.

[DETIENNE, 98a], DETIENNE V., *Structures relationnelles de données temporelles*, Projet TimeStamp, Institut d'Informatique, Facultés Universitaires de Namur, Novembre 1998.

[DETIENNE, 98b], DETIENNE V., *Conception physique d'une base de données temporelles en SQL-92 Eléments méthodologiques*, Projet TimeStamp, Institut d'Informatique, Facultés Universitaires de Namur, Novembre 1998.

[DETIENNE, 98c], DETIENNE V., *Structures de données bitemporelles*, Etude de cas, Projet TimeStamp, Institut d'Informatique, Facultés Universitaires de Namur, Décembre 1999.

[HAINAUT, 2000], HAINAUT J-L., Bases de données et modèles de calcul, Dunod, 2000.

[HAINAUT, 99], HAINAUT J-L., *Les bases de données temporelles, Introduction pratique*, Programme DB-MAIN, Projet TimeStamp, Institut d'Informatique, Facultés Universitaires de Namur, Juin 1999.

[JENSEN, 99], JENSEN Christian S., SNODGRASS Richard T., *Temporal Data Management*, IEEE Transactions on Knowledge and data engineering, VOL. 11, NO. 1, pp.36-44 January/February 1999.

[KAKOUDAKIS, 96a], KAKOUDAKIS I., THEODOULIDIS B., *The TAU Temporal Object Model*, Technical report, UMIST, Computation Departement, October 1996.

[KAKOUDAKIS, 96b], KAKOUDAKIS I., THEODOULIDIS B., *The TAU Time Model*, Technical report, UMIST, Computation Departement, October 1996.

[KARVELIS, 94], KARVELIS G., *An Algebra and a Query Language for the Entity Relationship Time Data Model*, thesis, UMIST, Computation Departement,1994.

[NORTH, 95], NORTH K., *Windows Multi-DBMS programming*, John Wiley & Sons, Inc, 1995.

[SNODGRASS, 2000], SNODGRASS Richard T., *Developing Time-Oriented Database Applications in SQL*, Morgan Kaufmann Publishers, 2000.

[SNODGRASS, 95], SNODGRASS Richard T., *The TSQL2 Temporal query language*, Kluwer Academic Publishers, 1995.

[THIRAN, 98], THIRAN P., DEMOULIN O., *ODBC*, Document interne, Projet Inter-DB, Institut d'Informatique, Facultés Universitaires de Namur, 1998.

Annexes

Table des matières

Annexe A :

Application de l'algorithme de coalescing à deux phases 1

Annexe B :

Application de l'algorithme de coalescing à une phase 7

Annexe C :

Les différentes cellules composant l'arbre d'une requête 13

Annexe D :

Le dictionnaire temporel 15

Annexe E :

Windows Interactive TSQL 19

Annexe A : Application de l'algorithme de coalescing à deux phases

Cette annexe réalise le coalescing à deux phases sur la table `PERSONNE_4` lorsque nous procédons à la synthèse des colonnes `NOM`, `ADRESSE`, `V_DEBUT`, `V_FIN`, `T_DEBUT`, `T_FIN`.

Avant de procéder à la résolution des problèmes horizontaux, l'ensemble de la table doit être ordonnée selon l'identifiant d'entité (`NOM`), `T_DEBUT` et `V_DEBUT`. La table `PERSONNE_4` étant déjà ordonnée selon ces colonnes, nous pouvons directement procéder à l'élimination des problèmes horizontaux.

PERSONNE_4						
NOM	**ADRESSE***	**SALAIRE***	**V_DEBUT**	**V_FIN**	**T_DEBUT**	**T_FIN**
Léo	Namur	50000	10	999	20	30
Léo	Namur	60000	10	20	30	45
Léo	Liège	60000	20	25	30	40
Léo	Liège	70000	25	999	30	40
Léo	Namur	80000	20	30	40	55
Léo	Bruxelles	80000	30	999	40	50
Léo	Namur	70000	10	20	45	55
Léo	Bruxelles	100000	30	999	50	60
Léo	Liège	50000	10	30	55	60
Léo	Liège	50000	10	35	60	999
Léo	Bruxelles	100000	35	999	60	999
...	...	...	...	...	...	...

Phase 1 : résolution des problèmes horizontaux

1) Lecture de l'enregistrement : Léo Namur 10 999 20 30

Table de travail					
NOM	**ADRESSE***	**V_DEBUT**	**V_FIN**	**T_DEBUT**	**T_FIN**
Léo	Namur	10	999	20	30

Table finale					
NOM	**ADRESSE***	**V_DEBUT**	**V_FIN**	**T_DEBUT**	**T_FIN**
-	-	-	-	-	-

2) Lecture de l'enregistrement : Léo Namur 10 20 30 45

Table de travail					
NOM	**ADRESSE***	**V_DEBUT**	**V_FIN**	**T_DEBUT**	**T_FIN**
Léo	Namur	10	20	30	45

Table finale					
NOM	**ADRESSE***	**V_DEBUT**	**V_FIN**	**T_DEBUT**	**T_FIN**
Léo	Namur	10	999	20	30

3) Lecture de l'enregistrement : Léo Liège 20 25 30 40

Table de travail					
NOM	**ADRESSE***	**V_DEBUT**	**V_FIN**	**T_DEBUT**	**T_FIN**
Léo	Namur	10	20	30	45
Léo	Liège	20	25	30	40

Table finale					
NOM	**ADRESSE***	**V_DEBUT**	**V_FIN**	**T_DEBUT**	**T_FIN**
Léo	Namur	10	999	20	30

4) Lecture de l'enregistrement : Léo Liège 25 999 30 40

Table de travail					
NOM	**ADRESSE***	**V_DEBUT**	**V_FIN**	**T_DEBUT**	**T_FIN**
Léo	Namur	10	20	30	45
Léo	Liège	20	999	30	40

Table finale					
NOM	**ADRESSE***	**V_DEBUT**	**V_FIN**	**T_DEBUT**	**T_FIN**
Léo	Namur	10	999	20	30

5) Lecture de l'enregistrement : Léo Namur 20 30 40 55

Table de travail					
NOM	**ADRESSE***	**V_DEBUT**	**V_FIN**	**T_DEBUT**	**T_FIN**
Léo	Namur	10	20	30	45
Léo	Namur	10	30	40	45
Léo	Namur	20	30	45	55

Table finale					
NOM	**ADRESSE***	**V_DEBUT**	**V_FIN**	**T_DEBUT**	**T_FIN**
Léo	Namur	10	999	20	30
Léo	Liège	20	999	30	40

6) Lecture de l'enregistrement : Léo Bruxelles 30 999 40 50

Table de travail					
NOM	**ADRESSE***	**V_DEBUT**	**V_FIN**	**T_DEBUT**	**T_FIN**
Léo	Namur	10	30	40	45
Léo	Bruxelles	30	999	40	50
Léo	Namur	20	30	45	55

Table finale					
NOM	**ADRESSE***	**V_DEBUT**	**V_FIN**	**T_DEBUT**	**T_FIN**
Léo	Namur	10	999	20	30
Léo	Liège	20	999	30	40
Léo	Namur	10	20	30	40

7) Lecture de l'enregistrement : Léo Namur 10 20 45 55

Table de travail					
NOM	**ADRESSE***	**V_DEBUT**	**V_FIN**	**T_DEBUT**	**T_FIN**
Léo	Bruxelles	30	999	40	50
Léo	Namur	10	30	45	55

Table finale					
NOM	**ADRESSE***	**V_DEBUT**	**V_FIN**	**T_DEBUT**	**T_FIN**
Léo	Namur	10	999	20	30
Léo	Liège	20	999	30	40
Léo	Namur	10	20	30	40
Léo	Namur	10	30	40	45

8) Lecture de l'enregistrement : Léo Bruxelles 30 999 50 60

Table de travail					
NOM	**ADRESSE***	**V_DEBUT**	**V_FIN**	**T_DEBUT**	**T_FIN**
Léo	Namur	10	30	45	55
Léo	Bruxelles	30	999	50	60

Table finale					
NOM	**ADRESSE***	**V_DEBUT**	**V_FIN**	**T_DEBUT**	**T_FIN**
Léo	Namur	10	999	20	30
Léo	Liège	20	999	30	40
Léo	Namur	10	20	30	40
Léo	Namur	10	30	40	45
Léo	Bruxelles	30	999	40	50

9) Lecture de l'enregistrement : Léo Liège 10 30 55 60

Table de travail					
NOM	**ADRESSE***	**V_DEBUT**	**V_FIN**	**T_DEBUT**	**T_FIN**
Léo	Bruxelles	30	999	50	60
Léo	Liège	10	30	55	60

Table finale					
NOM	ADRESSE*	V_DEBUT	V_FIN	T_DEBUT	T_FIN
Léo	Namur	10	999	20	30
Léo	Liège	20	999	30	40
Léo	Namur	10	20	30	40
Léo	Namur	10	30	40	45
Léo	Bruxelles	30	999	40	50
Léo	Namur	10	30	45	55

10) Lecture de l'enregistrement : Léo Liège 10 35 60 999

Table de travail					
NOM	ADRESSE*	V_DEBUT	V_FIN	T_DEBUT	T_FIN
Léo	Liège	10	35	60	999

Table finale					
NOM	ADRESSE*	V_DEBUT	V_FIN	T_DEBUT	T_FIN
Léo	Namur	10	999	20	30
Léo	Liège	20	999	30	40
Léo	Namur	10	20	30	40
Léo	Namur	10	30	40	45
Léo	Bruxelles	30	999	40	50
Léo	Namur	10	30	45	55
Léo	Bruxelles	30	999	50	60
Léo	Liège	10	30	55	60

11) Lecture de l'enregistrement : Léo Bruxelles 35 999 60 999

Table de travail					
NOM	ADRESSE*	V_DEBUT	V_FIN	T_DEBUT	T_FIN
Léo	Liège	10	35	60	999
Léo	Bruxelles	35	999	60	999

Table finale					
NOM	ADRESSE*	V_DEBUT	V_FIN	T_DEBUT	T_FIN
Léo	Namur	10	999	20	30
Léo	Liège	20	999	30	40
Léo	Namur	10	20	30	40
Léo	Namur	10	30	40	45
Léo	Bruxelles	30	999	40	50
Léo	Namur	10	30	45	55
Léo	Bruxelles	30	999	50	60
Léo	Liège	10	30	55	60

Avant de procéder à la résolution des problèmes verticaux, nous devons vider la table de travail en transférant ses informations dans la table finale. Nous obtenons donc la table finale ci-dessous.

Table finale					
NOM	**ADRESSE***	**V_DEBUT**	**V_FIN**	**T_DEBUT**	**T_FIN**
Léo	Namur	10	999	20	30
Léo	Liège	20	999	30	40
Léo	Namur	10	20	30	40
Léo	Namur	10	30	40	45
Léo	Bruxelles	30	999	40	50
Léo	Namur	10	30	45	55
Léo	Bruxelles	30	999	50	60
Léo	Liège	10	30	55	60
Léo	Liège	10	35	60	999
Léo	Bruxelles	35	999	60	999

Phase 2 : résolution des problèmes verticaux

Avant de procéder à la résolution des problèmes verticaux, la table résultant de la résolution des problèmes horizontaux doit être ordonnée selon l'identifiant d'entité `(NOM)`, `V_DEBUT` et `T_DEBUT`.

Table finale					
NOM	**ADRESSE***	**V_DEBUT**	**V_FIN**	**T_DEBUT**	**T_FIN**
Léo	Namur	10	999	20	30
Léo	Namur	10	20	30	40
Léo	**Namur**	**10**	**30**	**40**	**45**
Léo	**Namur**	**10**	**30**	**45**	**55**
Léo	Liège	10	30	55	60
Léo	Liège	10	35	60	999
Léo	Liège	20	999	30	40
Léo	**Bruxelles**	**30**	**999**	**40**	**50**
Léo	**Bruxelles**	**30**	**999**	**50**	**60**
Léo	Bruxelles	35	999	60	999

Afin de bien comprendre le procédé, les informations à synthétiser sont mises en gras, ce qui donne pour résultat la table suivante.

Table finale					
NOM	**ADRESSE***	**V_DEBUT**	**V_FIN**	**T_DEBUT**	**T_FIN**
Léo	Namur	10	999	20	30
Léo	Namur	10	20	30	40
Léo	**Namur**	**10**	**30**	**40**	**55**
Léo	Liège	10	30	55	60
Léo	Liège	10	35	60	999
Léo	Liège	20	999	30	40
Léo	**Bruxelles**	**30**	**999**	**40**	**60**
Léo	Bruxelles	35	999	60	999

Annexe B : Application de l'algorithme de coalescing à une phase

Cette annexe réalise le coalescing à une phase sur la table `PERSONNE_4` lorsque nous procédons à la synthèse des colonnes `NOM`, `ADRESSE`, `V_DEBUT`, `V_FIN`, `T_DEBUT`, `T_FIN`.

Avant de procéder à la résolution des problèmes horizontaux, l'ensemble de la table doit être ordonnée selon l'identifiant d'entité (`NOM`), `T_DEBUT` et `V_DEBUT`. La table `PERSONNE_4` étant déjà ordonnée selon ces colonnes, nous pouvons directement procéder à l'élimination des problèmes horizontaux.

PERSONNE_4						
NOM	**ADRESSE***	**SALAIRE***	**V_DEBUT**	**V_FIN**	**T_DEBUT**	**T_FIN**
Léo	Namur	50000	10	999	20	30
Léo	Namur	60000	10	20	30	45
Léo	Liège	60000	20	25	30	40
Léo	Liège	70000	25	999	30	40
Léo	Namur	80000	20	30	40	55
Léo	Bruxelles	80000	30	999	40	50
Léo	Namur	70000	10	20	45	55
Léo	Bruxelles	100000	30	999	50	60
Léo	Liège	50000	10	30	55	60
Léo	Liège	50000	10	35	60	999
Léo	Bruxelles	100000	35	999	60	999
...	...	...	...	...	...	...

1) Lecture de l'enregistrement : Léo Namur 10 999 20 30

Table de travail 1					
NOM	**ADRESSE***	**V_DEBUT**	**V_FIN**	**T_DEBUT**	**T_FIN**
Léo	Namur	10	999	20	30

Table de travail 2					
NOM	**ADRESSE***	**V_DEBUT**	**V_FIN**	**T_DEBUT**	**T_FIN**
-	-	-	-	-	-

Table finale					
NOM	**ADRESSE***	**V_DEBUT**	**V_FIN**	**T_DEBUT**	**T_FIN**
-	-	-	-	-	-

2) Lecture de l'enregistrement : Léo Namur 10 20 30 45

Table de travail 1					
NOM	**ADRESSE***	**V_DEBUT**	**V_FIN**	**T_DEBUT**	**T_FIN**
Léo	Namur	10	20	30	45

Table de travail 2					
NOM	ADRESSE*	V_DEBUT	V_FIN	T_DEBUT	T_FIN
Léo	Namur	10	999	20	30

Table finale					
NOM	ADRESSE*	V_DEBUT	V_FIN	T_DEBUT	T_FIN
-	-	-	-	-	-

3) Lecture de l'enregistrement : Léo Liège 20 25 30 40

Table de travail 1					
NOM	ADRESSE*	V_DEBUT	V_FIN	T_DEBUT	T_FIN
Léo	Namur	10	20	30	45
Léo	Liège	20	25	30	40

Table de travail 2					
NOM	ADRESSE*	V_DEBUT	V_FIN	T_DEBUT	T_FIN
Léo	Namur	10	999	20	30

Table finale					
NOM	ADRESSE*	V_DEBUT	V_FIN	T_DEBUT	T_FIN
-	-	-	-	-	-

4) Lecture de l'enregistrement : Léo Liège 25 999 30 40

Table de travail 1					
NOM	ADRESSE*	V_DEBUT	V_FIN	T_DEBUT	T_FIN
Léo	Namur	10	20	30	45
Léo	Liège	20	999	30	40

Table de travail 2					
NOM	ADRESSE*	V_DEBUT	V_FIN	T_DEBUT	T_FIN
Léo	Namur	10	999	20	30

Table finale					
NOM	ADRESSE*	V_DEBUT	V_FIN	T_DEBUT	T_FIN
-	-	-	-	-	-

5) Lecture de l'enregistrement : Léo Namur 20 30 40 55

Table de travail 1					
NOM	ADRESSE*	V_DEBUT	V_FIN	T_DEBUT	T_FIN
Léo	Namur	10	20	30	45
Léo	Namur	10	30	40	45
Léo	Namur	20	30	45	55

Table de travail 2					
NOM	ADRESSE*	V_DEBUT	V_FIN	T_DEBUT	T_FIN
Léo	Liège	20	999	30	40

Table finale					
NOM	ADRESSE*	V_DEBUT	V_FIN	T_DEBUT	T_FIN
Léo	Namur	10	999	20	30

6) Lecture de l'enregistrement : Léo Bruxelles 30 999 40 50

Table de travail 1					
NOM	ADRESSE*	V_DEBUT	V_FIN	T_DEBUT	T_FIN
Léo	Namur	10	30	40	45
Léo	Bruxelles	30	999	40	50
Léo	Namur	20	30	45	55

Table de travail 2					
NOM	ADRESSE*	V_DEBUT	V_FIN	T_DEBUT	T_FIN
Léo	Liège	20	999	30	40
Léo	Namur	10	20	30	40

Table finale					
NOM	ADRESSE*	V_DEBUT	V_FIN	T_DEBUT	T_FIN
Léo-	Namur	10	999	20	30

7) Lecture de l'enregistrement : Léo Namur 10 20 45 55

Table de travail 1					
NOM	ADRESSE*	V_DEBUT	V_FIN	T_DEBUT	T_FIN
Léo	Bruxelles	30	999	40	50
Léo	Namur	10	30	45	55

Table de travail 2					
NOM	ADRESSE*	V_DEBUT	V_FIN	T_DEBUT	T_FIN
Léo	Namur	10	30	40	45

Table finale					
NOM	ADRESSE*	V_DEBUT	V_FIN	T_DEBUT	T_FIN
Léo	Namur	10	999	20	30
Léo	Namur	10	20	30	40
Léo	Liège	20	999	30	40

8) Lecture de l'enregistrement : Léo Bruxelles 30 999 50 60

Table de travail 1					
NOM	ADRESSE*	V_DEBUT	V_FIN	T_DEBUT	T_FIN
Léo	Namur	10	30	45	55
Léo	Bruxelles	30	999	50	60

Table de travail 2					
NOM	ADRESSE*	V_DEBUT	V_FIN	T_DEBUT	T_FIN
Léo	Namur	10	30	40	45
Léo	Bruxelles	30	999	40	50

Table finale					
NOM	ADRESSE*	V_DEBUT	V_FIN	T_DEBUT	T_FIN
Léo	Namur	10	999	20	30
Léo	Namur	10	20	30	40
Léo	Liège	20	999	30	40

9) Lecture de l'enregistrement : Léo Liège 10 30 55 60

Table de travail 1					
NOM	**ADRESSE***	**V_DEBUT**	**V_FIN**	**T_DEBUT**	**T_FIN**
Léo	Bruxelles	30	999	50	60
Léo	Liège	10	30	55	60

Table de travail 2					
NOM	**ADRESSE***	**V_DEBUT**	**V_FIN**	**T_DEBUT**	**T_FIN**
Léo	Namur	10	30	40	55
Léo	Bruxelles	30	999	40	50

Table finale					
NOM	**ADRESSE***	**V_DEBUT**	**V_FIN**	**T_DEBUT**	**T_FIN**
Léo	Namur	10	999	20	30
Léo	Namur	10	20	30	40
Léo	Liège	20	999	30	40

10) Lecture de l'enregistrement : Léo Liège 10 35 60 999

Table de travail 1					
NOM	**ADRESSE***	**V_DEBUT**	**V_FIN**	**T_DEBUT**	**T_FIN**
Léo	Liège	10	35	60	999

Table de travail 2					
NOM	**ADRESSE***	**V_DEBUT**	**V_FIN**	**T_DEBUT**	**T_FIN**
Léo	Bruxelles	30	999	40	60
Léo	Liège	10	30	55	60

Table finale					
NOM	**ADRESSE***	**V_DEBUT**	**V_FIN**	**T_DEBUT**	**T_FIN**
Léo	Namur	10	999	20	30
Léo	Namur	10	20	30	40
Léo	Namur	10	30	40	55
Léo	Liège	20	999	30	40

11) Lecture de l'enregistrement : Léo Bruxelles 35 999 60 999

Table de travail 1					
NOM	**ADRESSE***	**V_DEBUT**	**V_FIN**	**T_DEBUT**	**T_FIN**
Léo	Liège	10	35	60	999
Léo	Bruxelles	35	999	60	999

Table de travail 2					
NOM	**ADRESSE***	**V_DEBUT**	**V_FIN**	**T_DEBUT**	**T_FIN**
Léo	Bruxelles	30	999	40	60
Léo	Liège	10	30	55	60

Table finale					
NOM	**ADRESSE***	**V_DEBUT**	**V_FIN**	**T_DEBUT**	**T_FIN**
Léo	Namur	10	999	20	30
Léo	Namur	10	20	30	40
Léo	Namur	10	30	40	55
Léo	Liège	20	999	30	40

Pour obtenir le résultat final, les deux tables de travail doivent être vidées. Cependant, comme certaines synthèses peuvent encore s'opérer, l'ensemble des éléments de la table de travail 1 doit être supprimé avant les éléments de la table de travail 2.

Les éléments de la table de travail 1 sont supprimés :

Table de travail 1					
NOM	**ADRESSE***	**V_DEBUT**	**V_FIN**	**T_DEBUT**	**T_FIN**
-	-	-	-	-	-

Table de travail 2					
NOM	**ADRESSE***	**V_DEBUT**	**V_FIN**	**T_DEBUT**	**T_FIN**
Léo	Liège	10	35	60	999
Léo	Bruxelles	35	999	60	999

Table finale					
NOM	**ADRESSE***	**V_DEBUT**	**V_FIN**	**T_DEBUT**	**T_FIN**
Léo	Namur	10	999	20	30
Léo	Namur	10	20	30	40
Léo	Namur	10	30	40	55
Léo	Liège	10	30	55	60
Léo	Liège	20	999	30	40
Léo	Bruxelles	30	999	40	60

Les éléments de la table de travail 2 sont supprimés :

Table de travail 1					
NOM	**ADRESSE***	**V_DEBUT**	**V_FIN**	**T_DEBUT**	**T_FIN**
-	-	-	-	-	-

Table de travail 2					
NOM	**ADRESSE***	**V_DEBUT**	**V_FIN**	**T_DEBUT**	**T_FIN**
-	-	-	-	-	-

Table finale					
NOM	**ADRESSE***	**V_DEBUT**	**V_FIN**	**T_DEBUT**	**T_FIN**
Léo	Namur	10	999	20	30
Léo	Namur	10	20	30	40
Léo	Namur	10	30	40	55
Léo	Liège	10	30	55	60
Léo	Liège	10	35	60	999
Léo	Liège	20	999	30	40
Léo	Bruxelles	30	999	40	60
Léo	Bruxelles	35	999	60	999

Annexe C : Les différentes cellules composant l'arbre d'une requête

Type de cellule	Utilisation du champ valeur	Cardinalité des fils	Types des fils
query	non	2-4	select (obligatoire) from (obligatoire) whereT (facultatif) whereNT (facultatif)
select	le champ valeur permet de stoker les informations sur la demande d'un « coalescing » et d'un « snapshot » .	1-N	token
from	non	1-N	table
table	le champ valeur permet de stocker le type de la table.	1-1	token
whereT	non	1-1	and or equals contains precedes meets overlaps starts finishes
whereNT	non	1-N	token
and	non	2-2	and or equals contains precedes meets overlaps starts finishes equal less_eq greater_eq
or	non	2-2	and or equals contains precedes meets overlaps starts finishes equal less_eq greater_eq

Type de cellule	Utilisation du champ valeur	Cardinalité des fils	Types des fils
equals	non	2-2	period timepoint transaction valid
overlaps	non	2-2	period timepoint transaction valid
starts	non	2-2	period timepoint transaction valid
finishes	non	2-2	period timepoint transaction valid
meets	non	2-2	period timepoint transaction valid
contains	non	2-2	period timepoint transaction valid
precedes	non	2-2	period timepoint transaction valid
transaction	non	1-1	token
valid	non	1-1	token
period	non	2-2	int0-999
timepoint	non	1-1	int0-999 token(« now »)
equal	non	2-2	int0-999
less_eq	non	2-2	int0-999
greater_eq	non	2-2	int0-999
token	le champ valeur est occupé par un string	0-0	
int0-999	le champ valeur est occupé par un entier compris entre [0-998]	0-0	

Annexe D : Le dictionnaire temporel

La table RDB$_TEMP_TABLE

Description :

RDB$TEMP_TABLE est la méta-table enregistrant l'ensemble des informations concernant les tables logiques. Grâce à cette méta-table, nous connaissons pour chaque table logique la transformation qu'elle a subie lors de l'optimisation, son type, son état et l'ensemble des tables physiques à accéder lorsqu'une requête temporelle est formulée.

Requête SQL :

```
CREATE TABLE RDB$TEMP_TABLE (
        TNAME VARCHAR(50),
        TRANSF SMALLINT,
        TYPE VARCHAR(2),
        STATE VARCHAR(1),
        TP_NOTCUR VARCHAR(50),
        TP_CUR VARCHAR(50));
```

Informations supplémentaires :

Les valeurs de la colonne TRANSF sont soit 1, 2 ou 3.

Les valeurs de la colonne TYPE sont :

MV (table mono-temporelle avec « valid time »),
MT (table mono-temporelle avec « transaction time »),
B (table bitemporelle).

Les valeurs de la colonne STATE sont :

n (table normalisée),
nn (table non normalisée).

La colonne TP_NOTCUR contient le nom de la table physique à accéder en cas de requêtes temporelles sur des données n'appartenant pas uniquement au présent.

La colonne TP_CUR contient le nom de la table physique à accéder en cas de requêtes temporelles sur des données appartenant uniquement au présent.

La table RDB$_TEMP_COLUMN

Description :

La table RDB$TEMP_COLUMN contient toutes les informations relatives aux colonnes d'une table temporelle. Chaque colonne est ainsi décrite par le nom de la table logique à laquelle il appartient, son numéro d'ordre, son nom, son type et deux booléens. Ceux-ci nous renseignent sur la possibilité d'avoir des données nulles et sur le caractère temporel de la colonne.

Requête SQL :

```
CREATE TABLE RDB$TEMP_COLUMN (
        TNAME VARCHAR(50),
        CSEQ SMALLINT,
        CNAME VARCHAR(50),
        CTYPE VARCHAR(20),
        LEN1 SMALLINT,
        LEN2 SMALLINT,
        NULLS CHAR(1),
        TEMP SMALLINT);
```

Informations supplémentaire :

Les colonnes CTYPE, LEN1 et LEN2 permettent de stocker le type d'une colonne de manière précise. Les différentes valeurs de ces colonnes sont :

CTYPE	LEN1	LEN2
CHAR	taille	-
DATE	-	-
DECIMAL	taille	décimal
FLOAT	-	-
INTEGER	-	-
NUMERIC	taille	décimal
SMALLINT	-	-
VARCHAR	taille	-

La table RDB$TEMP_KEY

Description :

La table RDB$TEMP_KEY permet de stocker les informations concernant les clés d'entité et les clés étrangères temporelles des différentes tables temporelles. Pour chaque colonne d'une clé, nous stockons les informations suivantes : le nom de la table d'où provient cette colonne, le type ainsi que l'identifiant de la clé dont elle fait partie et son nom. Par contre, s'il s'agit d'une clé étrangère, nous enregistrons les informations supplémentaires suivantes : le nom de la table et le nom de la colonne référencées.

Requête SQL :

```
CREATE TABLE RDB$TEMP_KEY (
        TNAME VARCHAR(50) NOT NULL,
        KTYPE CHAR(1) NOT NULL,
        KEYID VARCHAR(4) NOT NULL,
        CNAME VARCHAR(50) NOT NULL,
        TARGTAB VARCHAR(50),
        TARGCOL VARCHAR(50));
```

Informations supplémentaire :

Les valeurs de la colonne KTYPE sont :

e (clé d'entité),
f (clé étrangère).

La table RDB$TEMP_KEY

Description :

La table RDB$TEMP_KEY permet de stocker la valeur du temps présent.

Requête SQL :

```
CREATE TABLE RDB$TEMP_NOW (
        NOW INTEGER);
```

Annexe E : Windows Interactive TSQL

Ce programme permet de réaliser une requête mini-TSQL2 de manière interactive. Cependant, afin de montrer toutes les possibilités de flexibilité permises par les fonctions de TODBC, nous travaillerons avec des fonctions temporelles de TODBC particulières. Les hypothèses prises dans le cadre de ce programme sont nombreuses.

- Le littéral `snapshot` est éliminé de notre langage mini-TSQL2. En effet, toutes les requêtes de sélection incorporeront dans leur résultat les périodes adéquates. Cependant, celles-ci peuvent n'être liées (SQLBindCol) à aucune variable dans le programme, ce qui reviendrait à l'utilisation du littéral `snapshot`.

- Le littéral `every` est éliminé de notre langage mini-TSQL2. Celui-ci est remplacé par un nouvel argument dans notre fonction TSQLExecDirect. Cet argument de type booléen aura pour valeur par défaut la réalisation du coalescing.

- TSQLExecDirect fournit le nombre de colonnes du résultat de la sélection.

- TSQLExecDirect fournit le type du résultat de la sélection : non temporel (0), mono-temporel avec "valid time" (1), mono-temporel avec "transaction time" (2), bitemporel (3).

- TSQLExecDirect possède un nouvel argument qui récupère le nom, fournit par l'utilisateur, de la table résultant du coalescing.

- TSQLFreeStmt possède un nouvel argument qui récupère le nom, fournit par l'utilisateur, de la table résultant du coalescing.

```
//-----------------------------------------------------------------------------
// Windows Interactive TSQL
//-----------------------------------------------------------------------------

#include <owl\pch.h>
#include <owl\applicat.h>
#include <owl\framewin.h>
#include <owl\dialog.h>
#include <owl\static.h>
#include <owl\edit.h>
#include <owl\button.h>
#include <owl\menu.h>
#include <owl\checkbox.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include "wtsql.h"

//include for ODBC
# include <windows.h>
# include <sql.h>
# include <sqlext.h>

//include for TODBC
# include "C:\todbc\todbc.h"

SQLCHAR sqlchar[50];

HENV henv;
HDBC hdbc;
HSTMT hstmt;
RETCODE rc;
bool Isconnected;

//****************************************************************
//  ODBC3Error : gestion des erreurs des fonctions normales
//****************************************************************

void ODBC3Error(SWORD fHandleType, SQLHANDLE handle,char* comd)
{
 SQLINTEGER  fNative;
 SQLSMALLINT cb, psMsgNum;
 SQLCHAR     *szMessage;
 SQLCHAR     *szSQLState;

 szMessage = (SQLCHAR *) malloc(SQL_MAX_MESSAGE_LENGTH+1);
 szSQLState = (SQLCHAR *) malloc(6);
 psMsgNum = 1;

 SQLGetDiagRec (fHandleType, handle, psMsgNum, szSQLState, &fNative, szMessage,
                SQL_MAX_MESSAGE_LENGTH-1, &cb);

 MessageBox(NULL,szMessage,comd,MB_OK);
 MessageBox(NULL,szSQLState, "SQLSTATE",MB_OK);
 free(szMessage);
 free(szSQLState);
};
```

```
//*********************************************************************
//  TODBC3Error : gestion des erreurs des nouvelles fonctions de TODBC
//*********************************************************************

void TODBC3Error (char* comd)
{
 SQLCHAR      *szMessage;
 SQLCHAR      *szSQLState;
 RETCODE      rc;

 szMessage  = (SQLCHAR *) malloc(SQL_MAX_MESSAGE_LENGTH+1);
 szSQLState = (SQLCHAR *) malloc(6);

 rc= TSQLError(szSQLState, szMessage);

 if (rc == SQL_SUCCESS)
 {
  MessageBox(NULL,szMessage,comd,MB_OK);
  MessageBox(NULL,szSQLState, "SQLSTATE",MB_OK);
 };

 free(szMessage);
 free(szSQLState);
};

//*********************************************************************
//  Connect : réalise la connexion sur une base de données
//*********************************************************************

int Connect(char* DataName, char* User, char* Pass)
{
 RETCODE rc;

rc=SQLAllocHandle(SQL_HANDLE_ENV,SQL_NULL_HANDLE,&henv);
   if (rc==SQL_ERROR || rc==SQL_SUCCESS_WITH_INFO)
    {
     return 1;
    };
rc=SQLSetEnvAttr(henv,SQL_ATTR_ODBC_VERSION,(void *)SQL_OV_ODBC2,0);
   if (rc==SQL_ERROR || rc==SQL_SUCCESS_WITH_INFO)
    {
     ODBC3Error(SQL_HANDLE_ENV,henv,"SQLSetEnvAttr");
     return 1;
    };
rc=SQLAllocHandle(SQL_HANDLE_DBC,henv,&hdbc);
   if (rc==SQL_ERROR || rc==SQL_SUCCESS_WITH_INFO)
    {
     ODBC3Error(SQL_HANDLE_ENV,henv,"SQLAllocHandle");
     SQLFreeHandle(SQL_HANDLE_ENV,henv);
     return 1;
    };
rc=SQLSetConnectOption(hdbc,SQL_ATTR_ODBC_CURSORS,SQL_CUR_USE_ODBC);
   if (rc==SQL_ERROR || rc==SQL_SUCCESS_WITH_INFO)
    {
     ODBC3Error(SQL_HANDLE_DBC,hdbc,"SQLSetConnectOption");
```

```
     SQLFreeHandle(SQL_HANDLE_DBC,hdbc);
     SQLFreeHandle(SQL_HANDLE_ENV,henv);
     return 1;
    };
rc=SQLConnect (hdbc,(UCHAR FAR *) DataName , SQL_NTS,(UCHAR FAR *) User,
               SQL_NTS,(UCHAR FAR *) Pass , SQL_NTS);
   if (rc==SQL_ERROR || rc==SQL_SUCCESS_WITH_INFO)
    {
     ODBC3Error(SQL_HANDLE_DBC,hdbc,"SQLConnect");
     SQLFreeHandle(SQL_HANDLE_DBC,hdbc);
     SQLFreeHandle(SQL_HANDLE_ENV,henv);
     return 1;
    };
return 0;
};

//************************************************************************
//    class ConnectWindow
//************************************************************************

class ConnectWindow : public TDialog {
   public:
    ConnectWindow( TWindow* parent);
    void connect();
    void cancel();
    TEdit* EditData;
    TEdit* EditUser;
    TEdit* EditPas;
   private:

   DECLARE_RESPONSE_TABLE(ConnectWindow);
};

DEFINE_RESPONSE_TABLE1(ConnectWindow,TWindow)
 EV_COMMAND(IDOK,connect),
 EV_COMMAND (IDCANCEL,cancel),
END_RESPONSE_TABLE;

ConnectWindow::ConnectWindow( TWindow* parent)
: TDialog(parent ,"DIALOG_CONNECT")
{
 EditData = new TEdit(this,ID_ED_Data);
 EditUser = new TEdit(this,ID_ED_USER);
 EditPas = new TEdit(this,ID_ED_PAS);
 TColor* color = new TColor(10);
 SetBkgndColor(*color);
}

void ConnectWindow::cancel()
{
  CloseWindow();
}
```

```
void ConnectWindow::connect()
{
 char DataName[30], User[30], Pass[30];
 EditData->GetSubText(DataName,0,30);
 EditUser->GetSubText(User,0,30);
 EditPas->GetSubText(Pass,0,30);
 if (Isconnected==false)
   {
    if(Connect(DataName,User,Pass)==0)
      {
       MessageBox("Connected","OK",MB_OK);
       Isconnected=true;
      }
   }
 else
  MessageBox("Already connected","ERROR",MB_OK);

 CloseWindow();
}

/*************************************************************************/
//   class TInfoWindow
/*************************************************************************/
class TInfoWindow : public TDialog {
   public:
    TInfoWindow (TWindow* parent);
    char  nbcol[5];
    char temp[50];
    bool snapshot;
    bool coal;
    void info();
    void cancel();
    TEdit* EditTemp;
    TCheckBox* CheckSnap;
    TCheckBox* CheckCoal;
   private:

   DECLARE_RESPONSE_TABLE(TInfoWindow);
};

DEFINE_RESPONSE_TABLE1(TInfoWindow,TWindow)
 EV_COMMAND(IDOK,info),
 EV_COMMAND (IDCANCEL,cancel),
END_RESPONSE_TABLE;

TInfoWindow::TInfoWindow( TWindow* parent /*, int resId*/)
: TDialog(parent ,"DIALOG_INFO")
{
 EditTemp = new TEdit(this,ID_ED_TEMP);
 CheckSnap = new TCheckBox(this,ID_CB_SNAP);
 CheckCoal = new TCheckBox(this,ID_CB_COAL);
 TColor* color = new TColor(10);
 SetBkgndColor(*color);
}
```

```
void TInfoWindow::cancel()
{
 strcpy(nbcol,"");
 strcpy(temp,"");
 snapshot=true;
 coal=true;
 CloseWindow();
}

void TInfoWindow::info()
{
 EditTemp->GetSubText(temp,0,50);
 if(CheckSnap->GetCheck()== BF_CHECKED)
  snapshot=true;
 else
  snapshot=false;
 if(CheckCoal->GetCheck()== BF_CHECKED)
  coal=true;
 else
  coal=false;
 CloseWindow();
}

/************************************************************************/
//      class TWtsqlWindow
/************************************************************************/

class TWtsqlWindow : public TWindow {
  public:
    TWtsqlWindow();
    void run();

  private:
    void CmdFileConnect();
    void CmdFileDisconnect();
    void CmdFileExit();
    void CmdHelpStat();
    void CmdHelpAbout();
    TEdit* EditIn;
    TEdit* EditOut;
    TButton* ButtonRun;

  DECLARE_RESPONSE_TABLE(TWtsqlWindow);
};

DEFINE_RESPONSE_TABLE1(TWtsqlWindow,TWindow)
 EV_COMMAND(ID_BUT_RUN, run),
 EV_COMMAND(CM_CONNECT,CmdFileConnect),
 EV_COMMAND(CM_DISCONNECT,CmdFileDisconnect),
 EV_COMMAND(CM_EXIT,CmdFileExit),
 EV_COMMAND(CM_STATHELP,CmdHelpStat),
 EV_COMMAND(CM_ABOUT,CmdHelpAbout),
END_RESPONSE_TABLE;
```

```
TWtsqlWindow::TWtsqlWindow()
: TWindow (0,0,0)
{
 EditIn = new TEdit(this,ID_ED_IN,"",10,40,570,100,0,true);
 ButtonRun = new TButton(this, ID_BUT_RUN,"Run",240,152,75,25,false);
 EditOut = new TEdit(this,ID_ED_OUT,"",10,205,570,150,0,true);
 new TStatic(this,-1,"TSQL Statement",10,18,125,20,20);
 new TStatic(this,-1,"Output",10,185,100,20,20);
 TColor* color = new TColor(10);
 SetBkgndColor(*color);
 Isconnected=false;
}

void TWtsqlWindow:: run()
{
 //  unsigned char** liststring;
 int** liststring;
 SQLINTEGER* listcb;

 char query[200];
 char output[1000];
 int i,nbcol,type,nbbind;
 TInfoWindow* Info;

 Info = new TInfoWindow(this);
 Info->Execute();

 EditIn->GetSubText(query,0,200);


 rc=TSQLAllocStmt(hdbc,&hstmt);
   if (rc==SQL_ERROR || rc==SQL_SUCCESS_WITH_INFO)
    {
     TODBC3Error("TSQLAllocStmt");
     SQLDisconnect(hdbc);
     SQLFreeHandle(SQL_HANDLE_DBC,hdbc);
     SQLFreeHandle(SQL_HANDLE_ENV,henv);
     return;
    };

 rc=TSQLExecDirect(hdbc,hstmt,Info->temp,query,nbcol,type,Info->coal);
   if (rc==SQL_ERROR || rc==SQL_SUCCESS_WITH_INFO)
    {
     TODBC3Error("TSQLExecDirect");
     TSQLFreeStmt(hdbc,hstmt,Info->temp);
     SQLDisconnect(hdbc);
     SQLFreeHandle(SQL_HANDLE_DBC,hdbc);
     SQLFreeHandle(SQL_HANDLE_ENV,henv);
     return;
    };

liststring = new unsigned char* [nbcol];
for (int i = 0; i < (nbcol); i++)
   liststring[i] = new unsigned char [50];
listcb = new SQLINTEGER[nbcol];
```

```
//calcul du nombre de colonnes à lier

if(Info->snapshot==false)
 nbbind=nbcol;
else
 {
  if(type==1 || type==2)
    nbbind=nbcol-2;
  if(type==3)
    nbbind=nbcol-4;
 };

i=0;
while(i<nbbind)
{
rc=SQLBindCol(hstmt,i+1,SQL_C_CHAR,liststring[i],50,&listcb[i]);
 if (rc==SQL_ERROR || rc==SQL_SUCCESS_WITH_INFO)
    {
     ODBC3Error(SQL_HANDLE_STMT,hstmt,"SQLBindCol");
     TSQLFreeStmt(hdbc,hstmt,Info->temp);
     SQLDisconnect(hdbc);
     SQLFreeHandle(SQL_HANDLE_DBC,hdbc);
     SQLFreeHandle(SQL_HANDLE_ENV,henv);
     return;
    };
i++;
};

do
{
 rc=SQLFetch(hstmt);
 if(rc==SQL_NO_DATA)  break;
 if(rc==SQL_ERROR || rc==SQL_SUCCESS_WITH_INFO)
  {
   ODBC3Error(SQL_HANDLE_STMT,hstmt,"SQLFetch");
   break;
  };

 strcpy(output,"");
 i=0;
 while (i<nbbind)
 {
 strcat(output,liststring[i]);
 strcat(output," ");
 i++;
 };
 strcat(output,"\r\n");

}while(rc!= SQL_NO_DATA);

EditOut->SetText(output);

delete [] liststring;
delete [] listcb;
```

```
rc=TSQLFreeStmt(hdbc,hstmt,Info->temp);
 if(rc==SQL_ERROR || rc==SQL_SUCCESS_WITH_INFO)
    {
      TODBC3Error("TSQLFreeStmt");
      SQLDisconnect(hdbc);
      SQLFreeHandle(SQL_HANDLE_DBC,hdbc);
      SQLFreeHandle(SQL_HANDLE_ENV,henv);
      return;
     };

}

void TWtsqlWindow::CmdFileConnect()
{
  ConnectWindow* Connect;
  Connect = new ConnectWindow(this);
  Connect->Execute();
}

void TWtsqlWindow::CmdFileDisconnect()
{
 if(Isconnected==true)
  {
   SQLDisconnect(hdbc);
   SQLFreeHandle(SQL_HANDLE_DBC,hdbc);
   SQLFreeHandle(SQL_HANDLE_ENV,henv);
   MessageBox("Disconnected","OK",MB_OK);
  }
 else
  MessageBox("No connection in use","ERROR",MB_OK);
}

void TWtsqlWindow::CmdFileExit()
{
 MessageBox("Not yet implemented","EXIT",MB_OK);
}

void TWtsqlWindow::CmdHelpStat()
{
 MessageBox("Not yet implemented","HELP",MB_OK);
}

void TWtsqlWindow::CmdHelpAbout()
{
 MessageBox("Not yet implemented","ABOUT",MB_OK);
}

//{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{
//   class TwtsqlApp
//{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{

class TWtsqlApp : public TApplication {
  public:
    TWtsqlApp() : TApplication() {};
    void InitMainWindow();
};
```

```
void  TWtsqlApp::InitMainWindow()
{
 TFrameWindow* frame = new TFrameWindow(0, "Windows TSQL",new TWtsqlWindow);
 SetMainWindow(frame);
 GetMainWindow()->AssignMenu(MN_COMMANDS);
}

int OwlMain(int /*argc*/, char* /*argv*/ [])
{
  return TWtsqlApp().Run();
}
```